THIRTIETH ANNUAL
PACIFIC NORTHWEST
SOFTWARE QUALITY CONFERENCE

PNSQC™

October 8-10, 2012

World Trade Center Portland
Portland, Oregon

Permission to copy without fee all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

PNSQC™ 2012 President's Welcome

This year's theme "Engineering Quality" was brought about from the old adage that you can't test quality into a product. Recently I was sitting down with a software quality leader who was rightfully frustrated that a software team was ignoring all the basic best practices of software engineering, including code reviews, prioritized requirements, etc. The mentality of just throwing the code over the wall was still too much in play. How fitting is this year's conference theme for that engineering team!

This year we are pleased to have Matt Heusser kick-off the conference. Matt is probably best known for his writing. He is a contributing editor for Software Test & Quality Assurance Magazine and his personal blog "Creative Chaos" is consistently highly-ranked among software writing. As our first keynote speaker, he will be sharing with us "A Brief History of the Quality Movement and What Software Should Do About It". Through this he combines an insightful introduction to the quality movements of the 20th century, the story of what happened when those ideas were implemented in software, the system of effects those ideas swim in, and more, along with a few winning ideas for you to try out yourself.

On Tuesday Dale Emery will kick us off with a presentation on "Testing Quality In". Dale enjoys helping people apply the agile values of communication, feedback, simplicity, courage, and respect as they apply to all areas of software development. He will discuss how great testers apply many skills including detecting ambiguity, imagining troublesome scenarios, and creating concrete examples of abstract requirements. By applying these skills early in a project, we can help detect and correct misunderstandings before they become code.

If you have been to our conference in the past, we have kept the format of multiple tracks you are already familiar with, and given you plenty of choices throughout the two-day technical program. We have also maintained the 45-minute presentation format with 10 minutes between sessions to ensure presenters and attendees have plenty of time to get settled and ready for the next topic.

There is also opportunity for informal networking. On Monday night we have a reception, which includes poster papers on the mezzanine level with hors d'oeuvres and beverages. On Tuesday night we will have the pleasure of collaborating with Rose City Software Process Improvement Network (SPIN) to present Michael "Doc" Norton as he discusses agile metrics. You will leave this talk with a toolkit of additional metrics that, coupled with velocity, give a better view of the project's overall health. As we have done in the past, the Monday and Tuesday lunches will be Deep Dive Birds of a Feather discussion groups.

Our conference is full of practical, useful, and valuable information. With everyone's help, software quality continues to move forward. I am glad you are here and hope to network with each of you as we strive to accomplish our mission – *enable knowledge exchange to produce higher quality software.*

Doug Reynolds, President, PNSQC™ 2012

TABLE OF CONTENTS

Welcome i

Conference-At-Glance viii

Board Members, Officers and Committee Chairs x

Additional Volunteers xi

PNSQC Call for Volunteers xii

Keynote Addresses

A Brief History of the Quality Movement and What Software Should Do About It 1
Matt Heusser

Testing Quality In 3
Dale Emery

Invited Speakers

SDTaaS (SW Development & Test as a Service): Why? How? 5
Suzanne Miller

Growing into Excellence; T-Shaped and Beyond 13
Michael "Doc" Norton

The Power of an Agile Mindset in Determining Quality 15
Linda Rising

The Art of Software Investigation 23
Ben Simo

The Experience is the Thing: Having User Experience Drive Testing .. 35
Peter Walen

Agile Track

Waterfall to Agile: Flipping the Switch 47
Bhushan Gupta

Seeing Software Quality through an Agile Lens 55
Rhea Stadick

Distributed Agile Teams Are Faster than a Speeding Bullet! * 63
Aaron Akzin and Sudha Sudunagunta

Automation Track

Lightweight Extensible Provisioning System to Stage Heterogeneous Multi-Machine Test Environments **71**
Vladimir Averkin, Muhammad Usman Janjua and Sunil Kutty

Automating Build Installs Using AutoIt **91**
Brian Yoss

Occupy Cross Platform Testing **103**
Shilpa Ranganathan and Julio Lins

State-Based Testing Using Dynamic Instrumentation – A Case Study **113**
Vijay Upadya

30 Years of Regression Testing: Past, Present and Future **119**
Jean Hartmann

Catch Your Own Bugs: Including all Engineers in the Automation Cycle **127**
Laura Bright

Are We Failing at User Interface Test Automation? **139**
Jim Holmes

Green Lantern Automation **147**
Sridevi Pinjala

Cost Effective Agile Test Practices and Test Automation Using Open Source Tools SpecFlow and White * **165**
Kavitha Naveen and Perry Hunter

Code Analysis Track

Code Coverage Isn't Quality, It Isn't Even Coverage **179**
Wayne Roseberry

Introducing Static Analysis in a Mature Codebase **191**
John Ruberto

Design Track

Sniffing Out User Experience Smells ***201***
Jose Cuevas

How Design Trade-Offs Influence Software Testing ***209***
Anil Chakravarthy, Pramod Sharma and Sudeep Das

A Case Study into Improving Network Device Driver Quality by Design **217**
Anjali Singhai Jain and Kevin Scott

Self-Verifying Data **227**
Douglas Hoffman

Engineering Quality Track

Shake 'n' Send: Enabling Feedback Submission Directly from Mobile Applications **237**
Billy Landowski and Sira Rao

Engineering Quality in the Robotic World **245**
Kingsum Chow, Vicki Niu, Ethan Takla, Ida Chow, MacLean Freed and Jeffery Wang

Management Track

An Action Model for Risk and Uncertainty in Decision Making – How to Avoid the Paralysis of Analysis **253**
Jeffrey A. Robinson

Cultural Intelligence: From Buzz Word to Biz Mark **263**
Valérie Berset-Price

Critical Factors Characterizing Projects and Lifecycle Models **291**
Kalman C Toth and Herman Migliore

Reduce QA Cost by Improving Productivity & Test Optimization **309**
Rajesh Kumar and Ajay K Chhokra

Metrics Track

Using Metrics to Drive Customer Satisfaction **331**
Jeff Fiebrich

Metric-Driven Fuzzing Strategy for Network Protocols **341**
Faraz Qadri, Vivek Jain and Aniket Malatpure

Process Improvement Track

On the Way to Meeting a Mandate: Transitioning to Large Scale Agile **357**
Aashish Vaidya, Catherine Row and Mark Jackson

Release Engineering: A Guideline for Successful Software Release **371**
Vadiraj Thayur and Jayashree Nagaraja

Speeding Up Cross-Platform Testing – A Case Study **381**
Jeffrey Weston

An Enterprise Framework for Evaluating and Improving Software Quality **393**
Philip Lew

Lightweight Software Process Assessment and Improvement **405**
Tom Feliz

Development-Driven Testing: Ensuring Testing Meets the Needs of Software Developers **425**
Tim Farley

Process Dynamics, Variations and Controls in Software Engineering * **439**
Shivanand Parappa

Test Track

Risk Management for the Real (and Imperfect) World **453**
Darryl Nicholson

Testing Site Personalization, Unpacking Personalization Sites through Emulation **463**
Amith Pulla

Testing in Production: It's Not Just for Services Anymore! **473**
Adil Gheewala, Sam Bedekar and Weifeng Yao

The Power of Process, People, and Tools when Testing a Complex Integration Landscape for a Very Large Initial Retail ERP Implementation **489**
Nicki Nicolo

The Black Swan of Software Testing – An Experience Report on Exploratory Testing **499**
Shaham Yusuf and Venkat Moncompu

Avoiding Overkill in Manual Regression Testing **517**
Lisa Shepard

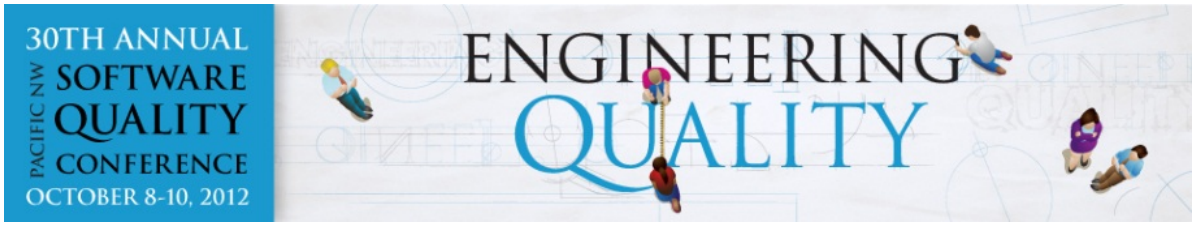
Get the Balance Right: Acceptance Test Driven Development, GUI Automation and Exploratory Testing * **531**
Michael Larsen

User Experience Track

Automated Testing: A Key Factor for Success in Video Game Development. Case Study and Lessons Learned **545**
Fazeel Gareeboo and Christian Buhl

Deriving Testing Strategies from User Experience Design Practices **561**
Venkat Moncompu

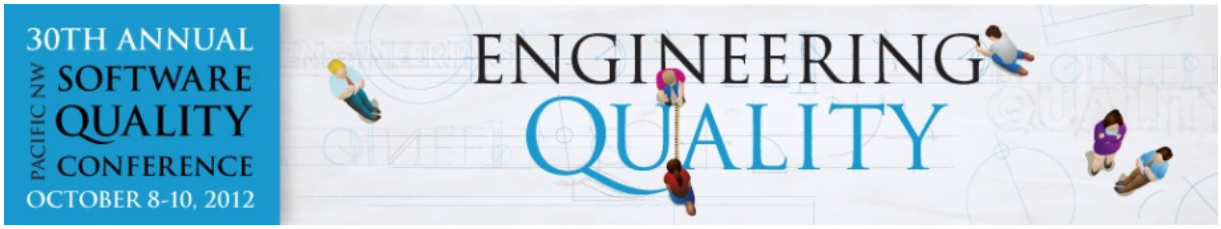
* These papers accepted by the Program Committee but not presented at the conference.



PNSQC 2012 – Conference At A Glance

Monday Oct 8, 2012 • Day One • Technical Program				
7:30-8:00 AM	**Registration and Exhibits Open**			
8:00-8:15 AM	**Welcome and Opening Remarks**			
8:15-9:45 AM	**KEYNOTE: A Brief History of the Quality Movement and What Software Should Do About It** Matt Heusser, Excelon Development, Page 1			
9:45-10:15 AM	**Break in Exhibit Hall**			
	INVITED SPEAKER	**CODE ANALYSIS**	**AUTOMATION**	**MANAGEMENT**
10:15-11:00 AM	**SDTaaS (SW Development & Test as a Service): Why? How?** Suzanne Miller, Software Engineering Institute (SEI) Page 5	Code Coverage Isn't Quality, It Isn't Even Coverage Wayne Roseberry, Microsoft Page 179	Lightweight Extensible Provisioning System to Stage Heterogeneous Multi-Machine Test Environments Vladimir Averkin, Microsoft Page 71	An Action Model for Risk and Uncertainty in Decision Making – How to Avoid the Paralysis of Analysis Jeffrey Robinson, AQI Page 253
11:10-11:55 AM		Introducing Static Analysis in a Mature Codebase John Ruberto, Intuit, Inc. Page 191	Automating Build Installs Using AutoIt Brian Yoss, McAfee Page 91	Cultural Intelligence: From Buzz Word to Biz Mark Valérie Berset-Price, Professional Passport Page 263
12:00-1:25 PM	**Lunch Program – Deep-Dive Discussions to Choose From**			
	INVITED SPEAKER	**Test**	**AUTOMATION**	**METRICS**
1:30-2:15 PM	**Growing into Excellence; T-Shaped and Beyond** Michael "Doc" Norton, LeanDog Page 13	Risk Measurement for the Real (and Imperfect) World Darryl Nicholson, Vesta Page 453	Occupy Cross Platform Testing Shilpa Ranganathan, Microsoft Page 103	Using Metrics to Drive Customer Satisfaction Jeff Fiebrich, Software Quality Page 331
2:25-3:10 PM		Testing Site Personalization, Unpacking Personalized Sites Through Emulation Amith Pulla, Intel Page 463	State Based Testing Using Dynamic Instrumentation – A Case Study Vijay Upadya, Microsoft Page 113	Metric-driven Fuzzing Strategy for Network Protocols Faraz Qadri, Microsoft Page.341
3:15-3:45 PM	**Break in Exhibit Hall**			
	INVITED SPEAKER	**TEST**	**Engineering Quality**	**MANAGEMENT**
3:45-4:30 PM	**The Power of an Agile Mindset in Determining Quality** Linda Rising, Consultant Page 15	Testing In Production: It's Not Just For Services Anymore! Adil Gheewala, Microsoft Page 473	Shake 'n' Send: Enabling Feedback Submission Directly From Mobile Applications Billy Landowski, Microsoft Page 237	Critical Factors Characterizing Projects and Lifecycle Models Kal Toth, Portland State University Page 291
4:40-5:20 PM		The Power of Process, People and Tools. . . Nicki Nicolo, Nike Page 489	Engineering Quality in the Robotic World Kingsum Chow, Intel Page 245	Reduce QA Cost By Improving Productivity & Test Optimization Rajesh Kumar, United Health Group Page 309

NOTE: Page numbers refer to the location within the Proceedings.



PNSQC 2012 – Conference At A Glance

Tuesday Oct 9, 2012 • Day Two • Technical Program				
7:30-8:00 AM	**Registration Open**			
8:00-9:30 AM	**KEYNOTE: Testing Quality In,** Dale Emery, Consultant, Page 3			
9:30-10:15 AM	**Break with Poster Papers**			
	INVITED SPEAKER	**TEST**	**DESIGN**	**PROCESS IMPROVEMENT**
10:15-11:00 AM	**The Art of Software Investigation** Ben Simo, QualityFrog Page 23	The Black Swan Of Software Testing – Experience Report On Exploratory Testing Shaham Yusuf, Deloitte Page 499	Sniffing Out User Experience Smells Jose Cuevas, Electronic Arts Page 201	On the Way to Meeting a Mandate: Transitioning to Large Scale Agile Aashish Vaidya, Cambia Health Solutions Page 357
11:10-11:55 AM		Avoiding Overkill in Manual Regression Tests Lisa Shepard, WebMD Page 517	How Design Trade-Offs Influence Software Testing Anil Chakravarthy, McAfee Page 209	Release Engineering: A Guideline for Successful Software Release Vadiraj Thayur, McAfee Page 371
12:00-1:25 PM	**Lunch Program – Deep-Dive Discussions to Choose From**			
	USER EXPERIENCE	**AUTOMATION**	**DESIGN**	**PROCESS IMPROVEMENT**
1:30-2:15 PM	Automated Testing: A Key Factor For Success in Video Game Development. Case Study and Lessons Learned Fazeel Gareeboo, EA Sports Page 545	30 Years of Regression Testing: Past, Present, and Future Jean Hartmann, Microsoft Page 119	A Case Study Into Improving Network Device Driver Quality By Design Anjali Jain, Intel Page 217	Speeding up Cross-Platform Testing – A Case Study Jeffrey Weston, Microsoft Page 381
2:25-3:10 PM	Deriving Testing Strategies From User Experience Design Practices Venkat Moncompu, West Monroe Partners Page 561	Catch Your Own Bugs: Including all Engineers in the Automation Cycle Laura Bright, McAfee Page 127	Self-Verifying Data Doug Hoffman, Software Quality Methods, LLC Page 227	An Enterprise Framework for Evaluating and Improving Software Quality Philip Lew, XBOSoft Page 393
3:15-3:45 PM	**Break with Poster Papers**			
	INVITED SPEAKER	**AUTOMATION**	**AGILE**	**PROCESS IMPROVEMENT**
3:45-4:30 PM	**The Experience is the Thing: Having User Experience Drive Testing** Peter Walen, Consultant Page 35	Are We Failing at User Interface Test Automation? Jim Holmes, Telerik Page 139	Waterfall to Agile: Flipping the Switch Bhushan Gupta, Nike Page 47	Lightweight Software Process Assessment and Improvement Tom Feliz, Tektronix Page 405
4:40-5:20 PM		Green Lantern Automation Sridevi Pinjala, IBM Page 147	Seeing Software Quality Through an Agile Lens Rhea Stadick, Intel Page 55	Development-Driven Testing: Ensuring Testing Meets the Needs of Software Developers Tim Farley, Gylity.com Page 425

NOTE: Page numbers refer to the location within the Proceedings.

PNSQC™ BOARD MEMBERS, OFFICERS, and COMMITTEE CHAIRS

Doug Reynolds – Board Member & President
Tektronix, Inc.

Les Grove – Board Member, Vice President & Program Co-Chair
Catalyst IT Services

Bill Baker – Board Member, Treasurer, Program Co-Chair & Operations Infrastructure Chair

Sion Heaney – Board Member & Community Outreach Chair

Bill Gilmore – Board Member & Marketing Communications Chair

Brian Gaudreau – Board Member & Audit Chair
Ciber, Inc.

Paul Dittman – Board Member & Program Co-Chair

Michael Dedolph – Invited Speaker Chair
Consultant

Shauna Gonzales – Conference Format Chair
Nike, Inc.

Srilu Balla – Volunteer Chair

PNSQC™ ADDITIONAL VOLUNTEERS

Aaron Akzin

Rick Anderson

Sue Bartlett

David Butt

Anand Chakravarty

Robert Cohn

Ian Dees

Moss Drake

Josh Eisenberg

Cynthia Gens

Daniel Grover

Leesa Hicks

Kathy Iberle

Michael Larsen

Cristina Marinovici

Gary Masnica

Launi Mead

Bill Opsal

Malathi Parthasarathy

Jeanette Schadler

David Socha

Keith Stobie

Deian Tabokov

Patt Thomasson

Timothy Thomasson

Mano Vela

Richard Vireday

Heather Wilcox

PNSQC™ Call for Volunteers

PNSQC is a non-profit organization managed by volunteers passionate about software quality. We need your help to meet our mission to enable knowledge exchange to produce higher quality software. Please step up and volunteer at PNSQC.

Benefits of Volunteering:

- Professional Development
- Contribution
- Recognition

Opportunities to Get Involved:

- **Program Committee** — Issues the annual Call for Technical Paper and Poster Paper Abstracts. Receives and manages the paper selection and review process and coordinates program layout.
- **Invited Speakers Committee** — Collaborates with the Program Committee to identify and invite leaders in the global quality community to provide conference speakers.
- **Marketing Communications Committee** — Ensures the software community is aware of PNSQC events via electronic and print media; coordinates and collaborates with co-sponsors and other organizations to get the word out. Identifies potential exhibitors and solicits their participation.
- **Operations & Infrastructure Committee** — Develops techniques to enhance communications with PNSQC board and committee members as well as the PNSQC community at large. Responsible for the PNSQC website, SharePoint, and speaker recording.
- **Community & Networking Committee** — Implements networking opportunities for the software community. Manages the social networking channels of communication. Recruits and works with incoming volunteers to place them in committees.
- **Conference Format Committee –** Provides programming for the networking opportunities at the conference. This includes the lunch time format and evening sessions. Responsibilities are recruitment of lead participants, compelling topics and titles and structure of the program.

Contact Us:

- On the *Survey Monkey* feedback form sent electronically to all PNSQC attendees immediately following the conference.
- At the website www.pnsqc.org, contact us link.

A Brief History of the Quality Movement and What Software Should Do About It

Matt Heusser
Excelon Development

From Frederick W. Taylor to Joseph Juran to Crosby and Deming, the 20th century is full of lessons about manufacturing quality.

A decade after that century ended we are left with some questions. Do those lessons still apply? If they do, which of them matter for software? Of the ones that matter, what experiments have we conducted to try to apply those lessons — and what can we learn from the results?

In this fast-paced talk Matt Heusser combines an insightful introduction to the quality movements of the 20th century, the story of what happened when those ideas were implemented in software, the system of effects those ideas swim in … and a few winning ideas to try on Monday.

Matthew Heusser *is a consulting software tester and software process naturalist who has spent his entire adult life developing, testing, or managing software projects. Along the way Matt has had the opportunity to serve as the lead organizer of the Great Lakes Software Excellence Conference (now in it's fifth year), lead organizer for the workshop on technical debt, and to teach information systems at night for Calvin College. Probably best known for his writing, A contributing editor for Software Test & Quality Assurance Magazine, Matt's personal blog "Creative Chaos" is consistently highly-ranked among software writing. An elected member of the Board of Directors of the Association for Software Testing, Matt recently served as lead editor for "How to Reduce the Cost of Software Testing" (Taylor and Francis, 2011). You can follow Matt on twitter at mheusser or email him matt.heusser@gmail.com.*

Testing Quality In

Dale Emery

Testing is an information service. It gives feedback about our work and our product. It helps us answer important questions, make informed decisions, correct mistakes and misunderstandings.

When we relegate testing to the end of the release cycle, we ensure that most of the feedback will be delivered late—after problems have been coded into the project, long after the actions that led to the results, and perhaps too late for us to act on. And if development slips, we often squeeze or cut testing in service to the release date. When we test only at the end, we deprive ourselves of information, and reduce the timeliness, relevance, and value of the information we deliver.

Great testers apply many skills: detecting ambiguity, imagining troublesome scenarios, and creating concrete examples of abstract requirements. By applying these skills early in a project, testers can help detect and correct misunderstandings before they become code. The tests created from these early conversations can guide development, gauge progress, and free our attention for other work. If we begin testing early, we can build quality into the product from the start.

Dale Emery *has worked in the software industry since 1980 as a developer, tester, manager, consultant, and trainer. In 2007 the Agile Alliance awarded Dale the Ward Cunningham Gentle Voice of Reason Award, which they created to recognize Dale's unique contribution to the Agile community. Dale helps people apply the Agile values of communication, feedback, simplicity, courage, and respect to all areas of software development.*

You can find Dale on his website, blog or on twitter at dhemery. He can also be found on Google+ and Linkedin

SDTaaS (SW Development & Test as a Service): Why? How?

Suzanne Miller – Software Engineering Institute (SEI)

In this presentation, Suzanne Miller, a researcher in both agile methods adoption and service delivery best practices, proposes a powerful mindset shift that is especially relevant for those who are adopting agile methods and looking for ways to refine their implementation. A key leverage point between services and agile methods is that both arenas focus on establishing and sustaining productive relationships. Successful service businesses (consulting, engineering services, health care) create relationships with customers that sometimes last decades, and evolve their offerings around client needs. Agile development organizations seek to create relationships with their product owners and end users that are sustained beyond a single delivery project, and when successful, their offerings change to meet the evolving needs of their clients. What can agile developers and testers learn from successful practices in service delivery? How does adopting a service mindset, and service practices, change the way agile projects are approached? This presentation will overview major changes in focus and practice that are possible. Ideas in this presentation are expanded in the tutorial later in the week, "Agile Methods + Services Best Practices: The Next Quality Advantage?"

Suzanne Miller *of the Software Engineering Institute at Carnegie Mellon University is best known for her modeling and process improvement research, including her book "CMMI Survival Guide: Just Enough Process Improvement". Her most recent research has been as part of a team exploring the use of lean and agile methods in DoD acquisition programs, and working with system of systems risk taxonomies. She also supports the CMMI for Services model by designing and delivering introductory and advanced training. She spent three years as the US Deployments Manager for a small commercial software firm, aimware. Prior to the SEI, Ms. Miller spent 12 years in a variety of technology transition roles at Lockheed Missile & Space Company. As for her education, she has a BS in Ergonomics from UC Santa Barbara and an MS in Systems Management from USC.*





This work was created in the performance of Federal Government Contract Number FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

This Presentation may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

NO WARRANTY

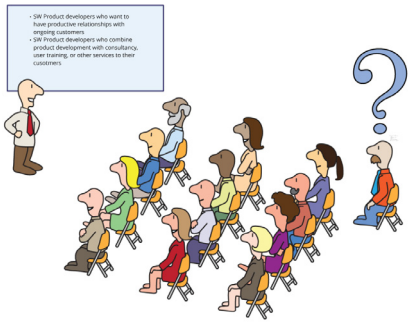
THIS MATERIAL OF CARNEGIE MELLON UNIVERSITY AND ITS SOFTWARE ENGINEERING INSTITUTE IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.



Purpose & Objectives

My purpose: to help you take advantage of service science concepts that provide high leverage for improving the performance of product development-focused organizations

copyright 2012, Carnegie Mellon University. All rights reserved.



Audience

- Different ways of looking at product development
- Non-development things a "customer intimate" product developer can do to improve performance
- Developers using agile methods are already on this path
- Summary

Agenda

What is a service?



Services are delivered through the operation of a service system.

Services imply ongoing relationships governed by agreements.

Services are often simultaneously produced and consumed.



Services have a different business rhythm than goods.

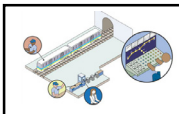


copyright 2012, Carnegie Mellon University. All rights reserved.

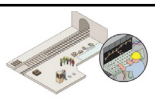
service: an intangible, non-storable product (e.g., operations, maintenance, logistics, and IT).

SuZ' favorite description of the key characteristic of a good service:

Co-Creation of Value





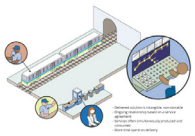


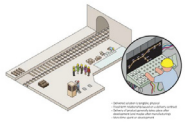
Services have a different business rhythm than goods.

Goods	Develop	Deliver
Service	Develop	Deliver

copyright 2012, Carnegie

How does service delivery differ from product development?





- Relationship often trumps content
- Attentiveness
- Immediacy and simultaneity
- Attention to details
- Consumers notice details immediately
- Visibility into how problems are resolved
- Real time communication
- Value strongly influenced by point of contact
- Quality can be objective or amorphous

- Iterating toward final correct product frequently expected
- Quality frequently defined in objective terms
- Value easily connected to specific performance results, independent of point of contact
- Communication infrequently real time
- Circuitous path from failure to producer

copyright 2012, Carnegie Mellon University. All rights reserved.

- Relationship often trumps content
- Attentiveness
- Immediacy and simultaneity
- Attention to details
- Consumers notice details immediately
- Visibility into how problems are resolved
- Real time communication
- Value strongly influenced by point of contact
- Quality can be objective or amorphous



- Delivered solution is tangible, physical
- Fixed-term relationship based on a delivery contract
- Delivery of product generally takes place after development (and maybe after manufacturing)
- More time spent on development

- Iterating toward final correct product frequently expected
- Quality frequently defined in objective terms
- Value easily connected to specific performance results, independent of point of contact
- Communication infrequently real time
- Circuitous path from failure to producer

copyright 2012 Carnegie Mellon University All rights r

How is product development becoming more similar to service delivery?

Similarities are most visible in an Agile environment....

- Relationship often trumps (initial) content
- Attentiveness, esp with customer collaboration
- Immediacy and simultaneity
- Attention to details
- Customers notice (software) details immediately
- Visibility into how problems are resolved
- Real time communication
- Value strongly influenced by point of contact
- Quality can be objective or amorphous

Why think about adopting a service mindset if you're a product developer?


Do we provide training services to others?


Do we provide sw maintenance services to others?


Do we provide analysis or engineering services to others?


Do we provide configuration or other logistics services to others?


Do we do provide test services to others?

●●●



We and our customers spend much of our time delivering services to others

copyright 2012, Carnegie Mellon University. All rights reserved.

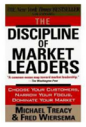
time delivering services to others

- **But are we providing them mindfully?**
- **Are we taking advantage of SERVICE concepts best practices to get and keep the customer relationships we need to have ongoing success?**

copyright 2012, Carnegie Mellon University. All rights reserved.

Setting the Context of Superior Service

The Discipline of Market Leaders



3 "disciplines" all successful businesses must be competent in:

- Operational excellence
- Product innovation
- Customer intimacy

BUT...

World class organizations excel at only one of these

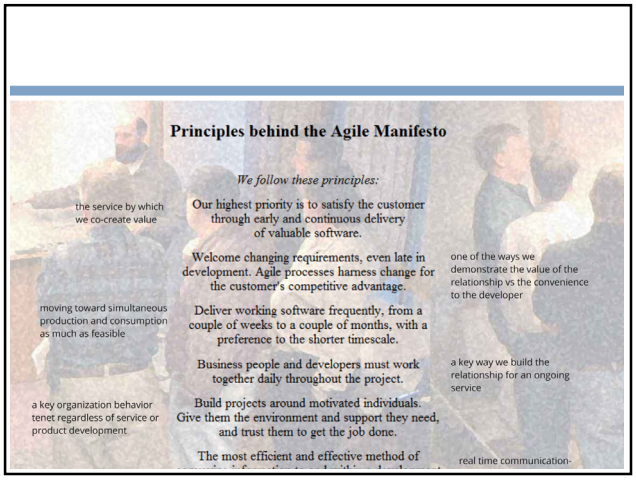
copyright 2012, Carnegie Mellon University. All rights reserved.

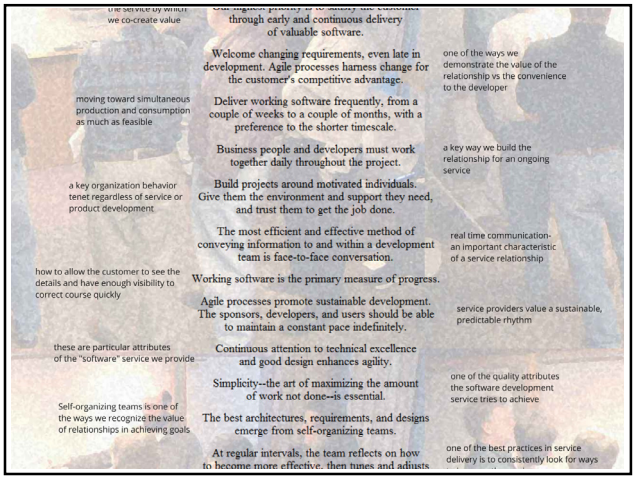
If, as an organization, you are trying to excel at (or even just be competent in) Customer Intimacy, you are engaging explicitly in building ongoing customer relationships -- you are adopting a service mindset toward your customers

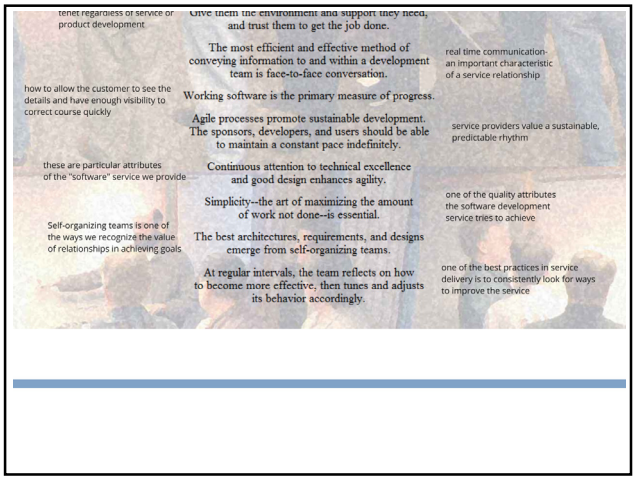
copyright 2012, Carnegie Mellon University. All rights reserved.

Most Agile Principles Directly Reflect a Service Mindset

copyright 2012, Carnegie Mellon University. All rights reserved.

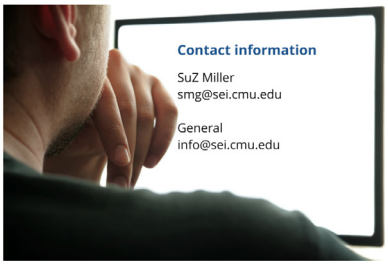






Summary: What do we do differently if we mindfully adopt a service mindset?

- Think about the "service agreement" we want with our customers (internal or external)
- Look at the things that can go wrong with the relationship as much as looking at what can go wrong with the product
- Look at and explicitly manage capacity and availability issues related to providing our software or test service
- Be mindful and explicit about the boundaries of our "service catalog": do we do or not do installation and user training, for example
- Be mindful of what it takes to transition our products into use and to evolve them (both from an architecture and a whole product viewpoint)
- Think about service continuity--how do we continue to provide our software or test service in the face of a significant disruption?
- Explicitly develop our service offerings, not just our product components



Contact information

SuZ Miller
smg@sei.cmu.edu

General
info@sei.cmu.edu

Growing into Excellence; T-Shaped and Beyond

Michael "Doc" Norton – LeanDog

We typically define our value to the company in terms of the role we fill; Business Analyst, Iteration Manager, QA Lead. But in today's high-paced, collaborative workplace, being good at just one thing is not likely enough to provide a life-long career. Those capable of filling other roles and being able to work well with others have a greater chance of longterm success. In this session we look at how we learn and grow into mastery, the value of broadening our skills and developing empathy for other roles, and finally take a look at some specific tools we can use to help succeed in a highly collaborative world.

A promoter and practitioner of agile since 1999, ***Michael "Doc" Norton*** *(@DocOnDev) is an agile coach and a partner with LeanDog. Doc's twenty-plus years of software development have provided him with experience on a wide range of topics. As a member of LeanDog, living in Wadsworth, OH, Doc provides coaching, mentoring, training, and delivery in agile/XP/lean software development techniques. Doc is immediately suspicious of anyone who declares expertise in a single programming language or methodology. He is passionate about helping others become better developers, working with teams to improve delivery, and emphasizing software craftsmanship.*

The Power of an Agile Mindset in Determining Quality

Linda Rising – Consultant

I've wondered for some time whether much of our success was the result of the placebo effect, that is, good things happened because we believed they would. The placebo effect is a startling reminder of the power our minds have over our perceived reality. Now cognitive scientists tell us that this is only a small part of what our minds can do. Research has identified what I like to call "an agile mindset," an attitude that equates failure and problems with opportunities for learning, a belief that we can all improve over time, that our abilities are not fixed but evolve with effort. What's surprising about this research is the impact of an agile mindset on quality, creativity and innovation, estimation, and collaboration in and out of the workplace. I'll relate what's known about this mindset and share some practical suggestions that can help all of us become aware of our own mindsets and how to improve them.

Linda Rising *has a Ph.D. from Arizona State University in the area of object-based design metrics. Her background includes university teaching as well as work in industry in telecommunications, avionics, and strategic weapons systems. She is an internationally known presenter on topics related to agile development, patterns, retrospectives, and the change process. Linda is the author of numerous articles and has published four books: Design Patterns in Communications, The Pattern Almanac 2000, A Patterns Handbook, and her latest book, written with Mary Lynn Manns, titled Fearless Change: Patterns for introducing new ideas. Her web site is: www.lindarising.org*

The Power of an Agile Mindset

Linda Rising
linda@lindarising.org
www.lindarising.org

Disclaimer: This provocative presentation is ideally the beginning of a conversation. It won't take long for me to tell you everything I know about cognitive psychology, although I have been reading in the area for several years now. I'm an amateur who has sufficient interest in weird topics and a strange way of connecting ideas that might or might not be of interest to you. Thank you for your tolerance and understanding of my meanderings and I hope you learn a little that might help you in your life.

This is not an "academic" presentation, but those interested in more information are invited to ask me for references for any part of this talk and I will be happy to make them available.

Do you mostly agree or mostly disagree with the following

(1) Intelligence is something very basic that you really can't change much. You're born with it or not. Yes, you can learn new things, but you can't really change how intelligent you are.

(2) No matter how intelligent you are, you can always get better, sometimes you can improve a lot.

Substitute any ability or talent for "intelligence."

Interesting experiments

Phase I:

Students were given a very easy set of questions

Then they were categorized into "effort" or "fixed" groups (about 50-50)

Phase II

All students allowed to choose between:
(1) a more difficult test (where they would learn a lot) or
(2) another easy test (similar to Phase I)

Most (~90%) "effort" kids chose (1)
Most (~80%) "fixed" kids chose (2)

Phase III

Very difficult exam given to both groups
"Effort" kids worked hard, enjoyed the challenge
"Fixed" kids easily discouraged

Phase IV

All students allowed to choose between:
(1) seeing exams of those who did better or
(2) exams of those who did worse

"Effort" kids chose (1)
"Fixed" kids chose (2)

Phase V

All students given easy test (similar to Phase I)
"Effort" kids improved (by ~30%)
"Fixed" kids' were worse (by ~20%)

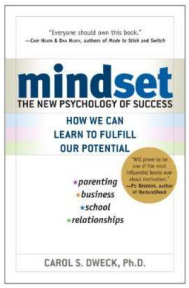
Phase VI

All students asked to give advice to other students and include their scores

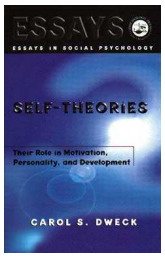
"Effort" kids: Lots of advice and encouragement.

"Fixed" kids: Very little or no advice and ~40% lied about their scores

Mindset – Carol Dweck



Self-Theories – Carol Dweck



Free on the web

New Yorker – "The Talent Myth," gladwell.com

New York Magazine – "How not to talk to your kids," Po Bronson

TIME magazine – "How to help them succeed"

Two mindsets: Fixed & Agile

Research shows that mindset:

Determines goals

Reactions to failure

Belief about effort and strategies

Attitudes toward others' successes

Two mindsets
Fixed vs. Agile

Ability – static, like height	Ability – can grow, like muscle
Goal - look good	Goal - to learn
Challenge - avoid	Challenge - embrace
Failure - defines your identity	Failure – provides information
Effort - for those with no talent	Effort - path to mastery
Reaction to challenge - helplessness	Reaction to challenge – resilience

Belief about yourself affects belief about others

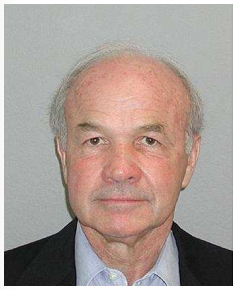
Those with a fixed mindset are quick to judge others based on little information and quick to stereotype

Those with an agile mindset don't ignore information about others but are less negative/positive

Bright little girls

Bright little boys

Smartest guys in the room



Enron's fixed mindset

Identify and hire "the best talent"

Then continuously and ruthlessly grade, sort, fire, and promote – a process called "rank and yank"

We are a company of people, not planes. That is what distinguishes us from other airlines.



The Southwest Culture

- Hire for attitude
- Establish a culture of community, trust, and the "Southwest spirit"
- Then provide learning opportunities and continually grow people
- Southwest seems to have an "agile" mindset

Managers have a mindset

"Pygmalion in Management," J. Sterling Livingston, Sept/Oct 1988 *Harvard Business Review*.

The good news is…

- Mindset is not "fixed"
- We encourage one or the other in each other
- We develop one or the other in our children
- Research has shown that small experimental manipulations, e.g. feedback or reading an article can produce one or the other
- Simply learning about the mindsets causes changes in people's belief systems so they are more likely to accept a growth mindset.

You mean I don't have to be stupid?

Effort is good!

Praise effort, strategies, process

Ask about the work

Instead of ignoring failure, teach others that it's a way to learn and improve

Agile software development

Fail early, fail often.

Fail fast, learn constantly.

Failure *IS* an option.

Without failure how can learning happen?

"Make mistakes faster."
Rich Sheridan, CEO, Menlo Innovations

"Perfect is a verb." Kent Beck

"Those that fail fastest grow strongest."
Roy Singham, Founder and Chair, Thoughtworks

Agile is agile

The agile mindset believes that we are *ALL* a work in progress.

The agile software development process isn't fixed. It continues to change and grow as we learn more about it. If we are lucky, this will never end because it will never be perfect.

We will continue to improve as we age—just like me ☺!

Ever tried. Ever failed. No matter. Try again. Fail again. Fail better. Samuel Beckett, Irish poet (1906-1989)

Thanks for listening!

The Art of Software Investigation

Ben Simo – QualityFrog

Process and tools play an important role in software testing, but let's not forget that the most important testing tool is the mind. Like scientists, testers are in the business of searching for new knowledge and sharing discoveries with others – hopefully for the betterment of people's lives. In both science and software, all discovery begins and ends in human intelligence.

Software testing is one of those cognitively complex activities that even those who practice it have difficulty describing. Those who try often give more credence to instruments than intelligence. If the mind is the most important part, let's not frame testing with processes and tools.

Over sixty years ago, William I.B. Beveridge reframed discussion of scientific research in his classic book *The Art of Scientific Investigation*. Rather than add to the many texts on the scientific method, he focused on the mind of the scientist. He shared principles and techniques for applying our minds to scientific research. His lessons apply to software testing today.

If we desire to discover and communicate new knowledge that matters, let's learn to think, and test, like scientists. Let's learn to use our most powerful testing tool and:

- Continually prepare
- Experiment
- Exploit chance
- Use hypothesis
- Imagine productively
- Apply intuition and reason
- Tune observation
- Overcome resistance

Ben Simo *is an amphibious, time-traveling, context-driven, cyborg software tester. Ben has been helping make quality assurance possible as a software tester and test tool developer for over 20 years. Ben has worked in a variety of industries, including: Defense, Healthcare, Finance, Education, and the Internet. Ben is a proponent of the Context-Driven School of software testing, and is the past President of the Association for Software Testing (AST). Ben blogs at QuestioningSoftware.com and IsThereAProblemHere.com. Ben tweets at QualityFrog*

THE ART OF SOFTWARE INVESTIGATION

Ben Simo
Ben@QualityFrog.com

PNSQC 2012

Based on
THE ART OF SCIENTIFIC INVESTIGATION
By W. I. B. Beveridge
1950

PREFACE

SOFTWARE TESTING

- What is it?

PREFACE

THE ART OF SCIENTIFIC INVESTIGATION

By William Ian Beardmore Beveridge

An entirely fresh approach
to the intellectual adventure
of scientific research

1950



PREFACE

COMPARING
SOFTWARE TESTING
TO SCIENTIFIC INVESTIGATION

CONTENTS

- PREFACE
- PREPARATION
- SELECTION
- SEQUENCE
- EXPERIMENTATION
- CHANCE
- HYPOTHESIS
- IMAGINATION
- INTUITION
- REASON
- OBSERVATION
- INVESTIGATORS
- DISCUSSION

Scientific research is not itself a science; it is still an art or craft.

- W. H. George

PREPARATION

SELF-DIRECTED LEARNING

- Knowledge
  - Build a foundation
  - Keep current
  - Maintain independence
  - Cultivate diversity
  - Understand history
- Fluency
  - Communicate & think with clarity
- Confer
  - Participate in the greater community

The research worker remains a student all his life. Preparation for his work is never finished for he has to keep abreast with the growth of knowledge.

- W. I. B. Beveridge

SELECTION

CHOOSE YOUR OWN WORK

- Interest encourages success
  - If work is chosen for you, seek out an aspect that provokes interest
- Select work that
  - has a chance of success
  - Is within your technical abilities

Start with a problem in which there is a good chance of his accomplishing something, and which is not beyond [your] technical capabilities.

- W. I. B. Beveridge

SEQUENCE

ITERATE

1. Review
2. Observe
3. Analyze
4. Guess
5. Experiment



The most effective experimenters are usually those who give much thought to the problem beforehand and resolve it into crucial questions and then give much thought to designing experiments to answer the questions.

- W. I. B. Beveridge

EXPERIMENTATION

TWO TYPES OF INVESTIGATION

- Observational
  - Collection of data from naturally occurring phenomena
- Experimental
  - Collection of data from an event made to occur under controlled conditions

All investigation is sampling

A basic concept … is that there is an infinitely large, hypothetical population of which the experimental group or data are a random sample.

- W. I. B. Beveridge

EXPERIMENTATION

EXECUTION

- Start modestly
  - Pilot
  - Sighting
  - Screening
- Take notes
  - Document as you go
- Iterate
  - Design later experiments based on results of earlier ones
- Stop
- Be competent
  - Techniques
  - Tools
  - Subject

It happens surprisingly often that one needs to refer back to some detail whose significance one did not realize when the experiment was carried out.

- W. I. B. Beveridge

EXPERIMENTATION

STATISTICS

- Caution
  - People give numbers more credence than they deserve
  - Averages are often misleading
  - Graphs are often misleading

The use of statistics does not lessen the necessity for using common sense in interpreting results, a point which is sometimes forgotten.

- W. I. B. Beveridge

EXPERIMENTATION

MISLEADING EXPERIMENTS

- Mistakes
  - "Honest" mistakes
  - Incompetent experimenters
- Contamination
  - Accidental or unknown influences
- Difficult to prove a negative

Experimentation, like other measures employed in research, is not infallible.

Inability to demonstrate a supposition experimentally does not prove that it is incorrect.

- W. I. B. Beveridge

EXPERIMENTATION

EUREKA

- Reproduce it
- Look at it from multiple perspectives
- Connect it with other knowledge
- Seek new avenues of investigation

The real and lasting pleasure in a discovery comes not so much from the accomplishment itself as from the possibility of using it as a stepping stone for fresh advances.

- W. I. B. Beveridge

CHANCE

THE ROLE OF CHANCE

- Chance plays an important part in discovery
  - Chance alone does not discover
  - Chance provides opportunity to the keen observer
  - Significance comes from an observer relating observations to other knowledge

In the field of observation, chance favors only the prepared mind.

- Pasteur

CHANCE

COURTING CHANCE

- Prepare your mind to recognize useful information
- Entertain ideas that contradict beliefs
- Be unconventional
- Maximize the risk of having a fortunate accident
- Postpone demand for evidence
- Perform many experiments

Chance favors only those who know how to court her.

- Charles Nicolle

CHANCE

RECOGNIZE & EXPLOIT

- Be alert for the unexpected
- Don't be blinded by hypothesis
  - Follow up on interesting side-issues

Acute powers of observation are often required to notice the clue, and especially the ability to remain alert and sensitive for the unexpected while watching for the expected.

- W. I. B. Beveridge

HYPOTHESIS

A TOOL FOR DISCOVERY

- Suggests new
  - Experiments
  - Observations
- Helps provide significance to what we observe
- Most will be wrong
  - Be prepared to abandon them

In science the primary duty of ideas is to be useful and interesting even more than to be 'true'.

- Wilfred Trotter

HYPOTHESIS

PRECAUTIONS

- Once an opinion is formed, it becomes difficult to think of alternatives
- Don't get too attached to your brainchild
- Let go of a hypothesis proved wrong

Men who have excessive faith in their theories or ideas are not only ill-prepared for making discoveries; they also make poor observations.

- Claude Bernard

HYPOTHESIS

SAFEGUARDS

- Subordinate ideas to facts
- Have multiple hypotheses
- Make special note of data unfavorable to your hypothesis
- Don't embrace conjecture
- Once the experiment begins, throw out the hypothesis

My business is to teach my aspirations to conform themselves to fact, not to try to make facts harmonize with my aspirations.

-Thomas Huxley

IMAGINATION

PRODUCTIVE THINKING

- Ideas "occur" to us
  - Can't deliberately create ideas
  - May come during
    - reflective thinking
    - daydreaming
- Fertilize your imagination
  - Variety of knowledge and experience
  - Focus thinking
  - Stay curious
- Temporarily suspend judgment
- Use reason to make ideas useful

To be genuinely thoughtful, we must be willing to sustain and protract that state of doubt which is the stimulus to thorough enquiry...

- Dewey

IMAGINATION

CAN BE DANGEROUS

- Don't repress it
  - Risk going astray
- Balance it
  - Criticism
  - Judgment
- Most hypotheses are wrong
  - Check your work
  - Detect and correct mistakes quickly

What merely annoys and discourages a person not accustomed to thinking ... is a stimulus and guide to the trained enquirer.

-Dewey

IMAGINATION

GETTING UNSTUCK

Temporary Abandonment

- Let it be
- Return once old thought associations are less strong
- Flaws in thinking become apparent

In research most of the time progress is difficult and often one is up against what appears to be a "brick wall".

- W. I. B. Beveridge

IMAGINATION

GETTING UNSTUCK

Discussion

- Useful suggestions
- Pooling information may trigger new ideas
- Detection of error
- Stimulating, refreshing
- Escape conditioned thinking
  - Explaining a problem requires clarifying information
  - Questioning by others disturbs our lines of thought

Productive mental effort is often helped by intellectual intercourse.

- W. I. B. Beveridge

INTUITION

SUDDEN ENLIGHTENMENT

- Arises from the subconscious
- Capture it

The really valuable factor is intuition.

- Albert Einstein

REASON

LIMITATIONS

- Logic has very little to do with discovery or invention
  - Logic builds on what is already thought to be so
  - Discovery often requires disregard for current beliefs

Great discoveries have been made by means of experiments devised with complete disregard for well accepted beliefs.

-W. I. B. Beveridge

REASON

SAFEGUARDS

- Don't confuse interpretation with results
  - Recognize that generalizations cannot be proved
  - Don't place excessive trust in generalizations

Research is fundamentally a state of mind involving continual re-examination of doctrines and axioms upon which current thought and action are based. It is, therefore, critical of existing practices

- Theobald Smith

REASON

INTERPRETING FACTS

- Group facts
- Identify relationships
- Assign significance
- Recognize consequences

Science consists in grouping facts so that general laws or conclusions may be drawn from them.

- Darwin

REASON

REASON SUPPORTS

- Judging ideas conjured up by imagination and intuition
- Planning experiments
- Deciding what observations to make
- Assessing the evidence
- Interpreting new facts
- Making generalizations
- Identifying applications of discoveries

Although discoveries originate more often from unexpected experimental results or observations, or from intuitions, than directly from logical thought, reason is the principle agent in most other aspects of research and the guide to most of our actions.

- W. I. B. Beveridge

OBSERVATION

EFFECTIVE OBSERVATION

1. Notice something
   - Things of interest
   - Changes in the familiar
2. Assign it meaning
   - Relating it to something else

What is observed depends on who is looking.

- W. H. George

OBSERVATION

DELIBERATE OBSERVATION

- Explicitly look for expectations
- Keep watch for the unexpected

Effective scientific observation also requires a good background, for only by being familiar with the usual can we notice something as being unusual or unexplained.

-W. I. B. Beveridge

PLANNING

LEVELS

- Tactical
  - Performed by the individuals doing the work
  - Short term
  - One experiment at a time
- Strategic
  - Performed by a larger group
  - Longer term
- Policy
  - Set priorities
  - Allocate resources

Discussions on planning research are often confused by failure to make clear what is meant by planning.

- W. I. B. Beveridge

PLANNING

IS NOT SCRIPTING

- Discovery is unforeseen
  - Infrequently comes from systematic accumulation of data
- Discovery requires
  1. Recognizing the unexpected
  2. Following it up
  3. Concentrated mental effort

The research worker ought not, having decided on a course of action, to put on mental blinders and, like a cart-horse, confine his attention to the road ahead and see nothing by the way.

- W. I. B. Beveridge

PLANNING

ALL PLANS ARE TENTATIVE

- Plan with an appropriate level of detail
- Adapt to discovery
- Communicate deviations from expectations

All plans must be regarded as tentative and subject to revision as the work progresses.

- W. I. B. Beveridge

INVESTIGATORS

ATTRIBUTES

- Pioneering attitude
  - Enterprising
  - Adventurous
  - Prepared for difficulty
  - Tenacious
  - Independent thinker
- Insatiable curiosity
  - Dissatisfaction with what is known
- Sometimes difficult
  - Lack confidence in their own views
  - Skeptical of others' views

The most successful scientists are capable of the zeal of the fanatic but are disciplined by objective judgment of their results and by the need to meet criticism from others.

- W. I. B. Beveridge

INVESTIGATORS

PREREQUISITES

- Willingness to work hard
- Intelligence
- Internal drive
- Imagination

It is not the talents we possess so much as the use we make of them that counts in the progress of the world.

- Brailsford Robertson

INVESTIGATORS

DISCOVERING DISCOVERERS

- Attributes of a good investigator are difficult to evaluate
- There is no exam
- Provide opportunity to demonstrate

Ordinary examinations are not a good guide to a student's ability at research, because they tend to favor the accumulators of knowledge rather than the thinkers.

- W. I. B. Beveridge

INVESTIGATORS

ETHICS

- Give credit to whom it is due
- Give generously
- Report sincerely
- Avoid secrecy

In the long run it pays the scientist to be honest, not only by not making false statements, but by giving full expression to facts that are opposed to his views.

- F. Cramer

DISCUSSION

Ben Simo
Ben@QualityFrog.com

The Experience is the Thing: Having User Experience Drive Testing

Peter Walen – Consultant

People using a computer system rarely use the features in a linear manner. They may use several related functions to do what they need to do: their "story" in the system. Other people will be using different functions at the same time. They will be engaged in their own stories. Why do so many testers and test groups ignore this simple idea? We may work using "stories" but do those stories reflect the real stories of the real users? If the users, our customers, are the ones who will accept or reject the software on its quality, can project teams and test groups adequately predict system behavior if our testing has little to do with the system user's behavior? Join Pete in a discussion of these ideas around broader function and integration testing with a focus on understanding true business needs and operations of the system.

Peter Walen *has been in software development for over 25 years. After working many years as a programmer he moved to software testing and QA. Following a brief foray in Project Management and Business Analysis, he returned to software testing. He has worked in the fields of Insurance and Finance, Manufacturing, Higher Education/Universities, Retail, Distribution and Point Of Sale Systems. Pete is an active member of several testing associations and an active blogger on software testing.*

Peter blogs at Rhythm of Testing and tweets at PeteWalen

The Experience is the Thing:

Having User Experience Drive Testing

PNSQC

Wednesday, October 9, 2012

User Experience Design

User Experience

Usability

© 2012, Peter G Walen

User Experience Design

Visual Design

Information Architecture

Interaction Design

Accessibility

Human-Computer Interaction

© 2012, Peter G Walen

User Experience

ISO 9241-210

Ergonomics of Human System Interaction

User Experience:

A person's perceptions and responses that result from the use or anticipated use of a product, system or service.

© 2012, Peter G Walen

Usability

User Friendly

Useable

"The ease of use and learnability of a human-made object."

© 2012, Peter G Walen

We have all read a bunch of stuff on this and seen and heard a bunch of speakers talk about how important this is.

© 2012, Peter G Walen

This presentation is NOT one of those.

© 2012, Peter G Walen

All of those ideas are important.

None of those ideas matter if the people using the software can't do what they need to do.

© 2012, Peter G Walen

Once Upon a Time…

A New System was being developed…

Distribution Center Management
Transportation/Delivery
Retail Item
Wholesale Item
Retail Pricing
Order Processing
Accounts Receivable
Accounts Payable

© 2012, Peter G Walen

Once Upon a Time…

New System developed to replace an "archaic" mainframe-based legacy system.

Loads of cool features implemented to help people do their jobs:
Auto-complete on Key-fields;
Windows/IE like Function Keys;
Screen Flow followed Printed Data Entry forms;

© 2012, Peter G Walen

Once Upon a Time…

They LOVED the new screens!

Very readable

Hover-over features

Way-easy to use compared to the mainframe system

© 2012, Peter G Walen

Once Upon a Time…

Everyone loved it

Except for one group

The people entering new items

© 2012, Peter G Walen

Once Upon a Time…

The data entry forms were used because "the users" were already familiar with them.

All the key fields were well known;
Thousands of copies were in circulation;
Customers & Data Entry staff were equally familiar with the forms and data;

© 2012, Peter G Walen

Once Upon a Time…

Follow the key-entry sheet…
Finish Page 1 and move to Page 2
Except an error message pops up:

Cannot save record:
Required field not populated

© 2012, Peter G Walen

Once Upon a Time…

Except the field is not on Page 1

Dialogue box pops-up to allow users to enter missing data

And processing continues…
until they press enter

© 2012, Peter G Walen

Once Upon a Time…

When an error message pops up:

Cannot save record:
Required field not populated

This message appears 5 times.

© 2012, Peter G Walen

Once Upon a Time…

How could this be?

The screens were developed from the data entry forms!

© 2012, Peter G Walen

Once Upon a Time…

How could this be?

The data entry forms were developed for use by the mainframe system being replaced.

© 2012, Peter G Walen

What Happened?

No one looked into how:

- the data entry form was arranged
  - the schema was defined
  - the old system worked

(and why)

© 2012, Peter G Walen

What Happened?

The Retail Item System was carefully tested.

The testers reported something "odd" and were told by the developers that it was what the design called for.

They asked if the user reps approved the design.

© 2012, Peter G Walen

What Happened?

The user reps were all from departments using the screens, and not people who entered new items.

They had no idea how other groups interacted with the system.

© 2012, Peter G Walen

What Happened?

Pages were formatted:

Without considering which fields were required;

For visual appeal in place of function;

© 2012, Peter G Walen

What Happened?

Moving from one page to the next included attempting to save the record being entered to the database.

Which did not allow nulls for five fields not on the first page.

© 2012, Peter G Walen

Lessons

No two "Users" have the same experience or interaction with a piece of software.

Sometimes the same person at two different times of the day has totally different interactions with the same software.

© 2012, Peter G Walen

Lessons

The best intentions around UX do not matter if the result is a system that *impedes* work instead of ***supports*** it.

© 2012, Peter G Walen

Lessons

Systematic Bias:

Tendency of a process to favor particular outcomes:
WAD Syndrome

If the people defining the rules are not the stakeholders of the business function, why are they defining the rules?

© 2012, Peter G Walen

Lessons

The behavior was noted by developers and testers. Development leadership wrote off the reports as Works as Designed.

Testers noted the behavior and failed to successfully advocate for the behavior to be addressed.

© 2012, Peter G Walen

One *MAJOR* idea was missed:

The "system" was not truly stand-alone!

The focus was on the **Retail Item System** – ***Other*** systems did not get the same detailed attention

© 2012, Peter G Walen

System of Systems

People using software don't see "*the system*" the same way most software people do.

People using software for different purposes will behave differently from each other.

© 2012, Peter G Walen

System of Systems

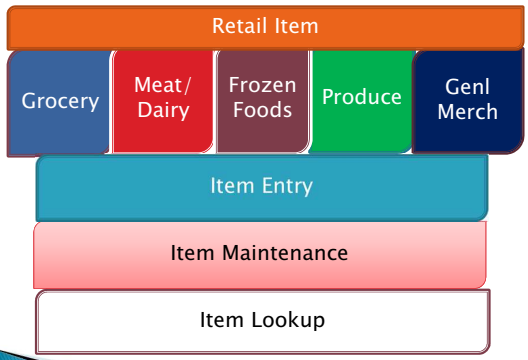
Their *Stories* are *Intertwined* but different.

© 2012, Peter G Walen

How Designers Viewed the System

Retail Item

© 2012, Peter G Walen



© 2012, Peter G Walen

Sometimes…

Small things can have devastating consequences.

© 2012, Peter G Walen

What if…

Can a difference in how data is displayed impact Usability?

How about User Experience?

Does Context matter here?

© 2012, Peter G Walen

Once Upon a Time…

System produced a screen display in one format.

Hard copy report in a similar, but slightly different format.

© 2012, Peter G Walen

Once Upon a Time…

Medical practitioner

In a hurry

Under pressure

Usually worked with one format

© 2012, Peter G Walen

Once Upon a Time…

One time Medical practitioner was working with the OTHER format.

They got confused.

© 2012, Peter G Walen

Once Upon a Time…

A patient was given the wrong dose of a medication.

The patient died.

© 2012, Peter G Walen

Lessons

Be aware of minor differences.

Be aware of human error.

Be aware that not everyone double checks everything every time.

© 2012, Peter G Walen

Contact Information

I can be reached here:

Twitter: @PeteWalen
Skype: Peter.Walen
Rhythmoftesting@blogspot.com

© 2012, Peter G Walen

Waterfall to Agile: Flipping the Switch

Bhushan Gupta
Nike Inc., Beaverton OR
bhushan.gupta@nike.com

Abstract

Agile software development methodology is transitioning from a fad to a practice. More and more software development practitioners are contemplating the switch from Waterfall to Agile; if not entirely, at least partially. Terms like scrum, sprint, stand up, story points, and velocity are becoming the *Lingua Franca*. While enough light has been shed on the mechanics of creating a release and iteration plan, managing story development, and sprint retrospectives, the team dynamics and behavior for success has not been given the mindshare it deserves.

Although the rules of the game, when moving from waterfall to agile development change substantially, the players and their goals do not. The player "product manager" now becomes "product owner" and needs to very actively pursue customer interest as a customer surrogate. Test engineers who took pride in breaking the code are now urged to work side by side with the developers and ensure that the defects are resolved immediately for a successful completion of a sprint. The project managers, who are supposed to drive the schedule, take on a role of facilitators making every effort to see that the sprint is complete and the velocity is adequate. Requirements that are frozen in the waterfall methodology rarely gel and the designs go through multiple refactoring cycles to make sure that the development progresses. The new product quality champions who were always looking for "the knee in the defect trends" now barely witness any fluctuations in the defect find trends from sprint to sprint. Refereeing the game, there is a new character in the theatre, "scrum master", who manages the delivery of the stories.

Can this change be achieved by simply creating a backlog, a release plan, and a sequence of iterations? No, it takes changes in people's perspectives, the way they think, and the way they achieve their goals. Would a Type A "product owner" who is used to taking a product to the market with all the bells and whistles be comfortable to do so with less than the planned functionality? Would a star developer forget about gold plating and would a test engineer who always took pride in finding errors in someone else's work be able to stand by developers and work with them? Most of all, will the orchestra be able to perform without the precisely written music; the "requirements document"? This paper analyzes the transition of a waterfall team to an agile environment and how changing the perspectives of the players can make the transition a success when the switch from Agile to Waterfall is flipped.

Biography

Bhushan Gupta has a M.S. in Computer Science from the New Mexico Institute of Mining and Technology, Socorro, NM. Bhushan has 27 years of experience in software engineering, 17 of which have been in the software industry. Currently a senior member of the GSS QA team at Nike, Bhushan is deeply involved with the agile development. Prior to joining Nike, Bhushan worked at Hewlett-Packard for 13 years in various capacities and led his groups in product development lifecycles, development methodology and execution processes, and software metrics for quality and software productivity. Bhushan has published and presented numerous articles in various conferences and has participated in panel discussions.

Copyright Bhushan Gupta August 15, 2012

1 Introduction

Prior to the 1990's, the software development practice involved converting well defined customer requirements into a product using procedural languages, following a sequence of milestones and checkpoints, and "freezing" the current state at a milestone to avoid last minute changes. This methodology, commonly known as the "waterfall lifecycle", included best practices such as "test first" where the test cases were developed from the requirements in parallel with the code development. The waterfall lifecycle focused on a high level of customer involvement during requirements definition and early prototyping but this involvement faded away as the product development proceeded. The customer was brought back into the game after the product was fully developed, this time to accept the product.

In the 1990's two major changes were brought into the world of software development; the procedural programming was replaced by Object Oriented programming and "Extreme Programming" was introduced as an alternative to the waterfall lifecycle (Wikipedia 2012). The "Extreme Programming," introduced by Kent Beck in 1996, provided a new vector to software lifecycle with focus on incremental development and frequent customer review to better suit the ever-changing customer requirements. This new approach also provided the ability to meet the 'First to Market" challenges presented by the Internet boom. The approach adopted some of the best practices, such as the test first of waterfall lifecycle. "Xtreme" (commonly spelled) programming morphed over time as new practices evolved and took on a common name "agile" development. Over the past decade a significant amount of change has taken place and the agile development is taking a hold. Shifting paradigm requires a change in mind set of the people involved. Is it happening? Are key people changing their behaviors as we move forward with the agile approach?

This article establishes the personality traits that are essential to build a successful agile team. Once internalized, the traits can be acquired by a team as it blazes the agile development trail. The article analyzes the role of each key team member, identifies the human characteristics that are relevant, and establishes a transition that facilitates an effective agile development environment.

2 Brief Background on Personality Types and Traits

Before we drill deeper into agile teams it is prudent to draw a distinction between personality type and traits. A personality type refers to quality of a person while a personality trait embodies a smaller grouping of behavioral tendencies. According to Type A and Type B personality theory, impatient, achievement-oriented people are classified as Type A, whereas easy-going and relaxed individuals are designated Type B (Wikipedia 2012). The other classification of personality based upon the behavior is as follows:

- Neuroticism – tendency to experience unpleasant emotions relatively easily,
- Extraversion – tendency to seek simulation and enjoy company of others,
- Conscientiousness – tendency to show self-discipline, consultative, competence, order, dutifulness, and thorough,
- Agreeableness – tendency to be compassionate towards others,
- Openness to Experience – tendency to enjoy new intellectual experiences and ideas, and
- Cognitive – abstract level thinking, analytical etc. (relatively new).

3 Key Players in Software Development

The software development involves multiple actors; a *customer*, *product owner*, *project manager*, *program manager*, *development team*, *quality assurance (QA) team*, *business systems analyst*, and a *technical writer*. An additional character in the agile development is the *scrum master* whose job is to orchestrate the product development rhythm. In the agile development method, the product manager is

normally called “product owner”. The scrum master role in the traditional waterfall environment was normally attributed to the project manager who made sure that the development was on track. There is a transformation these key actors need to make to support the change.

3.1 The Customer

Regardless of the development methodology, the customer plays an important role in the product development. Hayes (Hayes) has described four categories of the customers: analytical, driver, amiable, and expressive. In the case of agile development the objective is to create a high customer value and thus work very closely with the customer. Often times, a real customer is not available on site to work with the development and a surrogate customer fulfills this role.

An “analytical” customer can enhance the stories by bringing order, precision, discipline, and logic to the development, resulting in a better quality product. However, to achieve success iteration after iteration and maintain the rhythm, a “driver” type customer is very desirable. A “driver” personality is action oriented, decisive, demanding, forceful, competitive in nature, and will contribute to finish each iteration on time. The “driver” personality should be complemented with the expressive personality to bring in motivation, influence, confidence, and optimism. An amiable customer will be appreciative of the results achieved in each sprint. It will be highly desirable to build a small team of surrogate customers that is inclusive of all four customer personality types.

Although it is highly beneficial to have a customer with the product delivery team, it is hard to convince one to dedicate time with the team unless the customer is internal. In most cases, the product owner becomes the customer surrogate. As discussed below, the product owners are often Type A personalities. To fulfill the customer role they also need to be drivers, forceful, decisive, commanding, action oriented, and possess superior interpersonal skills. It is a challenge to find someone with all these traits. What is the solution? This can be achieved by a tag-team where the scrum master can contribute to the customer role provided he/she is comfortable in surrendering control to the product owner who ultimately answers to the customer.

3.2 Product Owner

Often a market savvy individual, the product owner has a keen sense of customer requirements and a goal to make future products more successful than the current in the market place. Driving the product qualities to delight customers, the product manager’s motto is to give the customer all the bells and whistles so that the product can take the customer by storm and easily beat the competition. A product owner, normally a Type A personality full with ambitions, rigidly organized, takes on more than she/he can handle. They are often a highly achieving individual, are proactive and obsessed with time management. These personality traits translate into a strong desire to yield a market winner product delivered on time.

In a typical waterfall environment, the product manager facilitates gathering requirements, making sure that the requirements are understood and agreed upon by all stakeholders, and are NOT subject to change. The requirements are said to be frozen at this stage. Once the requirements are frozen, the product manager often does not necessarily interact with the customer or actively looks for new requirements that may have risen due to time lapse.

Agile Product Owner blog (Agile Product Owner, 2012) explicitly lists the responsibilities of an agile product owner. When introduced to an agile development, a product manager, now the product owner often desires to have a complete picture of the product. If a product is being ported to a new platform, one of the motivations of this actor is to assure that the entire functionality is carried over to the new platform. There is a high probability that the product owner would insist on mapping the entire product functionality to epics and stories and gather details on each story to feel assured that no functionality is missing in the new product. Once the stories have evolved, the temptation is to develop the associated tasks and the acceptance criteria for each story. This leads to spending a large amount of time up front in gathering and

perfecting stories. It is equivalent to flipping back to the requirements gathering phase of the waterfall methodology. Now that all the stories have been defined, the product owner is ready to move to story prioritization.

The product owners have a difficult transition to make. For a product owner, it is important to understand that success in the agile development is achieved by perfecting one story at a time. They need to be cognizant of the fact that the product will grow over the entire development and what has been developed so far must provide value to the customer. Instead of nailing down all the epics, he/she needs focus on an epic that delivers the most value to the customer and builds customer interest. The product then evolves based upon the customer interest and the added value it provides. When porting a product to a new platform, it is important not to dwell into each epic and drill down every story just because the entire functionality is known. It is an opportunity to take an epic, analyze it from the customer perspective and enhance it to provide a superior customer experience. The product owner must focus on one story at a time.

3.3 Scrum Master

Scrum Master, the backbone of agile development, has a distinct personality. Peter Deyoe in his blog (Peter Deyoe, 2009) describes the following characteristics of a Scrum Master:

- Be humble enough to serve the team
- Have a strong character and be confident enough to stay in the background, promoting the team
- Display a high degree of integrity and maintain a trusting relationship
- Be politically savvy with a strong relationship with the Product Owner and,
- Be able to understand both business and technical people

“The best Scrum Masters are real team players, who receive as much satisfaction from facilitating others’ success as their own. They must also be comfortable surrendering control to the Product Owner and team. For those two reasons, traditional project managers don’t usually make great Scrum Masters (Scrum Methodology, 2010).”

The above statement could be controversial as the *project managers* do possess integrity and are trusting. They also work very closely with the product owner and have a good technical understanding of the product. The project managers are technical and thus can understand technical intricacies, dependencies, as well as help and support the team. Often the project managers have a good understanding of team capabilities and can help develop good sprint contents. However, a project manager, by the nature of his/her role, may not be able to stay in the background and relinquish control. Agile teams are self motivated and any attempt to control them can result into chaos.

Is it appropriate for a *program manager* to lead the scrum master role? A program manager oversees several related project aspects and is a strong driver of overall product delivery. Experience shows that the program manager often becomes a de facto Scrum Master. The nature of their job is not entirely different than that of a typical project manager and unless they are well trained they will run into the same problem as the project managers.

The program managers are strongly driven by the outcome and therefore maintain a sharp focus on schedule. By training, they are very conscientious of milestones and checkpoints. At the same time they are not accustomed to being in the background and are not current on the technical aspects. The role of the scrum master has been well defined and the scrum master skills can be acquired via formal training widely available. A well trained scrum master would make a team very successful.

3.4 Development Team

The development team in the current context refers to software architects, designers, and developers. These are the characters that are most impacted by change. Before we look into the change, let us first establish the traits of these professionals. By nature, software architects, designers, and developers are creative people. They are innovative, flexible, and are open to new experience and new ideas. They are always interested in improvements and like to discuss their ideas with their colleagues. Often under schedule pressure they are afraid of failure.

The software engineers have been measured as follows on these traits (Sodiya et al., 2007):

- Neuroticism: Low
- Extraversion: Medium
- Conscientiousness: Medium
- Openness To Experience: High
- Cognitive Capability: High
- Agreeableness : High

In a typical waterfall environment, the project is not only bounded, but the architectural, design, and coding activities are very structured. The scope is defined to the degree that the requirements are frozen and the probability of any change is very low. On the other hand, the stories in the agile environment are brief and the details are added in the form of tasks and acceptance criteria. Lack of details in the story is substantiated with on-going interactions between the stakeholders. There is more room for misinterpretations, misunderstandings, and missing requirements. Since the development is incremental in nature and the architecture and design elements evolve leading to refactoring, there is a higher degree of freedom to be creative and therefore a higher potential for chaos. In the waterfall model the discipline is achieved by milestones and checkpoints; in the agile methodology where the customer value is delivered every 2 or 3 weeks, self-discipline is very important. The nature of short delivery span enforces a high level of discipline, ongoing interaction with the stakeholders and on time delivery. Experience shows that successful agile teams always deliver the planned stories at the end of each sprint.

So, for a successful change from a waterfall to agile, a development team needs to posses strong cognition, extraversion, conscientiousness, and openness to experience traits. They need to learn how to deal with not-so-well defined requirements, continuously evolve the architecture and design, and be disciplined iteration after iteration after iteration. Vagueness in the deliverables though, provides opportunities to be creative but the short span of time in which the planned work needs to be delivered, requires making reasonably sound decisions. The nature of work changes form developing a product from the well defined requirements to rather loosely defined stories that need to be functional in a short span of time and deliver customer value. There are special needs that an architect faces to meet for a transition to be successful. Since the refactoring is a necessary evil in the agile development, an architect should have cognitive capability to be able to evolve the system architecture. It is necessary for the entire development team to possess openness to experience, extraversion, and agreeableness.

3.5 Quality Assurance (QA) Team

Studies have also measured the personality traits for software quality assurance professionals and the results are (Maverick Tester, 2010):

- Neuroticism: Low
- Extraversion: Medium
- Conscientiousness: Medium
- Openness To Experience: High
- Cognitive Capability: High
- Agreeableness : High

These traits are identical to the traits of the development team.

The Quality assurance team finds a high degree of change while moving from waterfall to agile. As discussed above, the requirements are brief and system behavior may not be well established except in the situations where the product is already in the market and going through enhancements. Often times, the story does not provide enough details and a significant amount of information is embedded in the acceptance criteria. Quality assurance team should not only review the stories and the acceptance criteria, it should also be on the lookout for any additional information that is relevant for testing. This requires extraversion, openness to experience, and agreeableness. The QA team must work very closely with the product owner and the business systems analyst to understand the functional requirements and derive the test cases with them on a continual basis; sprint after sprint. The test cases must be reviewed for each sprint. The environment requires a very close collaboration and not an “us versus them” mentality.

In a large project at Nike where the development is done by an external vendor, the QA team participates in information gathering sessions, communication exchanges, and story acceptance demos. The stories, after they are qualified by the vendor, are presented to the product owner for their approval to release to Nike QA team for final acceptance. The QA group has the liberty of contacting the vendor development team as often as necessary.

The QA team at Nike also works hand in hand with the development team. At first there is always resistance to direct communication from QA to developers, but soon both teams realize the criticality of direct and frequent communication between the two teams. At Nike, the development team continually encouraged the QA engineers to be actively involved with the development. As a result, the defect find, fix, and verification were expeditious in spite the fact that the QA team was not part of the development team. The defect resolution cycle was short and efficient. In another successful agile practice, a few members of the QA team were embedded into the development team. Whether the QA team is a part of the development team or not, its goal still remains the same – be a strong customer ally and a committed member of the project team.

The above discussion brings about the following points which are valuable for the transition from waterfall to agile:

1. The QA team does not have to be integrated with the development team and can still be effective and efficient.
2. A few members of the QA team can be embedded with the development team, temporarily reporting to the development (project) manager.
3. The team should be included in information exchange sessions and one-on-one discussions.

In agile development, the code is not “thrown over the wall” for testing. Successful teams will have a strong sense of inclusion regardless of the development methodology. Agile development environment forces this inclusion.

3.6 Build and Deployment Team

Often, build and deployment is a function of the development team. However, the responsibility is confined to a single or, at most, a few individuals. In the waterfall environment, build and deployment takes a big-bang approach where the code is delivered to the stakeholders towards the end. In the Agile environment the build and deployment happens in every sprint. There is a need to update the build and the opportunity to incrementally improve this process for accuracy and efficiency and finally, to perfection as the product gets closer to release.

The actors responsible for this activity require discipline to keep the process up to date and documented. In addition, the individuals should be proactively looking for the process improvement where the tools utilized to build and deploy are simple “like a single click script”. One should be evaluating the process at

each build and make changes to make it efficient. The two important personality traits for these individuals are extraversion and cognitive.

4 Making the Transition

The key pre-requisites to a making any transition are awareness, understanding, and commitment from the stakeholders (Bridges, 1991). Once these aspects of change are well understood and evaluated, then only a go - no go decision should be made. This understanding should include an evaluation of the project needs, team evaluation – personality traits, and training and coaching availability. Most successful agile teams start at a very small scale and go through an adaptation process. A project that is heavily dependent on the external partners may not be a good candidate for agile development. Ultimately, a project's success is measured on cost, schedule, and product quality. If flipping the switch does not achieve these success criteria or hampers any it is not the time for transition. Listed below are the steps that will facilitate the switch and ease the transition to agile:

4.1 Evaluate Potential and Need for Switch

- Assess if switching to agile reduces cost, shortens schedule, or improves quality of your product delivery
- Assess if the switch should be achieved via adaption over a multiple short projects or the team is ready to switch for a current complex project.

4.2 Evaluate Team Characteristics

- Let the team review personality traits of each role and assess the personal change that will be needed to make the switch
- Once change is identified, evaluate the commitment
- If the project involves an external (3rd party) vendor, assess if the vendor is capable to meet the requirements of change. To change an external vendor is beyond the team control.

4.3 Mentoring and Coaching

- Assess if the experienced mentors will be available to assist the team players as the team goes through the transition
- Prepare senior management to internalize the disruption caused by the change and the potential benefits of the change.

4.4 Measuring the Success

- Develop measures, not just the project progress, but the change in people behavior, project rhythm to understand the progress

4.5 Readiness

- Evaluate readiness before switching

5 Conclusion

Flipping a switch from Waterfall to Agile goes beyond the operational changes. It also includes careful evaluation of traits team members currently possess and the new traits they need to acquire. Even before considering a flip, the team needs to decide whether or not the switch will facilitate cost reduction, shorter schedule, and yet deliver the intended quality. Unlike waterfall, an agile development requires an interactive environment for the duration of the project. These interactions are very prominent between the *customer* and the *product owner*, *product owner* and *scrum master* as well as between the development and the QA teams. The personality traits of each player are critical for a successful transition to Agile.

References

Wikipedia, 2012, http://en.wikipedia.org/wiki/Extreme_programming

Wikipedia, 2012, http://en.wikipedia.org/wiki/Personality_type

Adam Hayes, “If You Only Understood Your Customer’s Personality Styles”, http://www.ahfx.net/weblog/37

Agile Product, Owner, LLC, http://agileproductowner.com/

Peter DeYoe, 2009, The Personality of a Great Scrum master, http://it-insight-blog.com/2009/10/the-personality-of-a-great-scrum-master/

Scrum Methodology, 2010, http://scrummethodology.com/the-scrummaster-role/

Sodiya. A.S., Onashoga, S.A., Longe, H. O. D., and Awodele O., An Improved Assessment of Personlaity Traits in Software Engineering, Interdisciplinary Journal of Information, Knowledge, and Management, Vol.2, 2007.

http://mavericktester.com/archive/personality-traits-in-software-testing/

Bridges, William, 1991, Managing Transitions, 3rd Edition, De capo Press, 1991, ISBN13 978-0-7382-1380-4Author-date style is usually used for references in the sciences.

Seeing Software Quality through an Agile Lens

Rhea Stadick
rhea.d.stadick@intel.com

Abstract

For a software quality engineer entering an Agile team for the first time, the transition can be extreme. The traditional software quality practices and activities change in many ways but, looking closely, the core principles are still there. In fact, Agile development can be a software quality engineer's dream! Agile naturally combines all factors that impact quality - from people to transparency - all in a cohesive system.

This paper focuses on the experience of one software quality engineer who went from a waterfall background and jumped head first into an Agile development environment. It details how Agile and specifically the Scrum process benefits quality. Additionally, it explains key concepts anyone starting Scrum should know and what they don't always tell you in the books. These are common "gotchas" when adopting Agile which will inevitably derail teams from achieving the quality promises associated with Agile product development if they're not understood. These topics will range from focus on technical workmanship and practices to accepting transformational change.

This paper will help quality professionals understand how Agile supports creating a quality organization and what to be aware of when embarking on an Agile journey.

Biography

Rhea Stadick is a software quality program manager for product engineering teams in the Business Client group at Intel. She is focused on developing a sustained, strong capability in engineering organizations to deliver high quality products. Over the last several years she has become an advocate for agile development and is an agile coach as well as a director of the Intel Agility Catalyst team promoting the adoption of agile culture and methodologies throughout the company. Within the Portland software community, she runs the Rose City Software Process Improvement Network (SPIN) which brings together software professionals in the area for networking and learning. Rhea graduated from Oregon State University with a Bachelor of Science Honors degree in Computer Science and holds an MBA from Willamette University.

Copyright Rhea Stadick 2012

1 Introduction

I began my software quality journey in a group that had years of experience in developing quality methodologies centered on a waterfall life cycle. This was due to our product life cycle being a staged-gate waterfall methodology. This had benefited our core hardware platforms which are expensive to change the further you move along the development timeline. However, we were developing software which is much cheaper to change. It is also higher in the stack which requires it to be at the mercy of any changes on dependent components in the system. In the complex environments that we operate in, not only the size of platforms but the complexity of the technology itself, confinement in a stage-gate life cycle that assumes you can cleanly exit one stage and not return simply did not work. I'd like to say I realized that the life cycle was a core issue in this process but when it surrounds and drives entire platforms, it's difficult to see what is really causing quality issues in the system. It's even more opaque when, as a quality engineer, you are focused on process improvement within the system, rather than looking at the system as a whole.

It wasn't until I joined a new software development team that was trying out something new (at least new to me) called "Agile" and specifically starting with an Agile project management process called Scrum. Immediately, I could see the benefits to quality in this environment: greater transparency into the quality, ability to change processes quickly, and focus on discipline within development. Compared to my experience in a waterfall life cycle, this process allowed me as a software quality engineer (SQE) to easily enforce quality while allowing rapid improvements when things weren't working. However, there were many things no one told me about what it took to actually be Agile and even how to really achieve a Scrum project methodology successfully. This paper details both the key benefits of adopting Scrum and transitioning to Agile as well as the key lessons I wish someone had told me from the very beginning. The key realization I've come upon in this seven year journey is that to achieve quality software in complex, emerging high-tech products, simply looking at life cycles or processes is not enough. It requires a full quality system to achieve, from the structure of organizations to the people and fundamentally to the culture of the development team. After having the benefit of working in two different approaches to product development, I've found Agile to be superior in enabling a system-focused view of quality that truly supports the fundamental goals of software quality professionals and ultimately of the business organization.

2 How Scrum Helps the SQE

2.1 Process Improvement

One of the biggest issues I faced in my role as a platform SQE was the very long feedback and improvement cycle. Only after we had left the requirements phase and got into development did we start to realize that there were problems in the requirements process. This could be months past the activity. Yet, due to life cycle and length of the programs, we couldn't improve the requirements process until the next platform. Months (if not over a year) on feedback loops is much too long for the fast moving high tech industry. It was also painful as an SQE to not be able to do much to fix an issue discovered in one phase that was root caused to an issue in a previous phase.

When I moved to the new team working with Scrum, the feedback loops become as long as the sprint; no more than four weeks. Our sprints happened to be two weeks at the time and eventually grew to three weeks. This single change was a complete transformation for me and the development team. We could identify issues in one sprint and change them the next and receive almost instantaneous feedback. Rather than wait entire programs for improvements, we waited a couple of weeks. Furthermore, Scrum as an empirical process builds in the required

inspect/adapt activities needed for continuous improvement. Pausing to see how the program was going and what needed to change to be successful wasn't an afterthought we performed in a post-mortem following the end of a project. It was something we did every three weeks as part of a key activity in the process – the sprint retrospective. Then, we could adjust and include any changes as we planned out the next sprint. This had a profound impact on quality by identifying any quality issues or impediments very quickly and having built-in regulation to fix them before they derailed the project.

2.2 Knowing Your Quality

The second key issue I faced when working on a waterfall life cycle was the lack of visibility into the quality of the software. Our process was set up so that the Alpha milestone meant "code complete" but not feature complete. This meant that while the team thought the software was done, there hadn't been enough validation to actually verify this. This meant that we wouldn't really have a good picture of where we were at until Beta which was near the end of the program! As an SQE, I would have to track what I could and cross my fingers that all the activities not yet performed to vet out the software wouldn't find a catastrophic defect that pushed out our schedule. This is no way to live! It was painful for SQEs, but even more painful for teams who had to suffer the long nights of fixing major issues found at Beta when there was little to no time left. And by the way, this was the first time customers were really seeing the full software so new requirements (which according to our stage-gate process should have been complete) were rolling in requiring more development. I won't go into the massive integrations that also would occur at these times which led to even more excitement and discovery of new and interesting issues.

Scrum is based on the Agile value of transparency. This is designed on one level by the empirical process but really shines with the requirement that user stories are "done done" by the end of every sprint. This doesn't mean "code complete" or "done with 5 critical defects." This means for all intents and purposes we feel the user story is complete and we could ship it to customers without biting our fingernails. From a quality perspective, that means at the end of each sprint I know the exact quality of the software that had been implemented so far. I would like to throw in a giant "but…" here and I would challenge many teams who claim they're following Scrum to say they have perfectly achieved this. It takes a lot of work and time to get to this level in a 3-week iteration. However, you can make significant strides towards this that get you to a state where you are confident in your released software quality and eliminate any last minute surprise defects. More on this later in the Lessons Learned section. Aside from the "but" sandwich, this was an incredible relief from a quality perspective. I was no longer waiting anxiously on these giant integration milestones where I was certain to find many new, hairy defects and show a sea of red in the quality criteria indicators. We had visible, working software every sprint and a clear understanding of any defects in the code.

2.3 Managing Complexity

We can't kid ourselves on how complex our projects are - from the technology, to the scope, to the human elements, and even the surrounding environment. We cannot plan everything and in fact when we start a project we know very little of the full requirements and how we'll implement the solution. Therefore, we must use a life cycle and more importantly create a system that enables us to work with uncertainty, adjust to changes at any point in the program, and give us mechanisms that allow us to see how the system is doing and quickly fix any broken elements. Ralph D. Stacey demonstrates in his book *Strategic Management and Organizational Dynamics, the Challenge of Complexity*, that as the understanding of technological implementation becomes more uncertain and requirements are far from agreement, the complexity of the project significantly increases. Waterfall life cycles are good for projects with lower complexity where technology is closer to certainty and there is a higher understanding of requirements. As we

move into projects with high complexity, it requires life cycles such as Scrum that are empirical rather than prescriptive and furthermore a larger systems-level approach to navigate the complexity.

At an even higher level, Agile focuses on the system level view of product development. It addresses the other complexities mentioned above such as organizational structure and team health. All quality professionals should know that product development does not just live in the space that we define by the life cycle and processes we put into place within that life cycle. In order to truly manage quality, we must have a way of addressing product development at a system level or we will inevitably be sidelined by something else in the system that we weren't addressing.

2.4 Team Ownership of Quality

The Agile values and 12 principles as described in the Agile Manifesto put a considerable amount of emphasis on high quality, working software delivered frequently. This focuses the entire team on owning quality and ensuring they are developing in a way that enables them to do this sustainably over time. On the Agile team, I was a ScrumMaster and by helping the team to follow the Agile principles and adhere to the Scrum process, I was effecting the same quality results as I was ultimately trying to achieve as an SQE. With the team owning quality, much better design, development, and testing practices emerged than I could have hoped to define on my own in the traditional SQE role. This was because the team was applying their expertise and defining ways to work better on a constant basis which resulted in much more advanced software development.

2.5 Quality is in the Eye of the Beholder

Last, but certainly not least, we get to the age old question of what is the definition of quality. Truth be told, we can define quality criteria all day long but we will never truly know unless we get it in front of a customer. Customer input and evaluation of software is just another feedback loop that is significantly shortened through the Scrum process. In waterfall we would develop all of the features and then throw them at the customer to get feedback. However, by definition of the life cycle, change was extremely hard so the likelihood of actually accepting customer feedback at that point was very low. Scrum allows and encourages you to get feedback from customers very early and frequently in the development life cycle. Because you have "done done" software in each sprint, you can now show them software that they can even evaluate. Furthermore, the process allows you to accept feedback at any point in the life cycle. At the end, you are much more assured that you have delivered a high quality product.

3 Embarking on the Journey: 5 Things You Need to Know

Based on the previous pages, you may be thinking that I've just drunk the kool-aid and believe that Scrum and Agile are a journey filled with rainbows and unicorns. Either that or *you've* drunk the kool-aid and want to jump right in and implement it tomorrow. Regardless of which side of the fence you're on, the truth is that actually applying Scrum is hard and you must implement other Agile methodologies to actually get to a state that resembles Scrum. Remember that Scrum is just a project management process. Many consultants don't talk about this but there's a lot more to do in order to get it right. So, for the skeptics and those that are about to jump in, here are the things I learned from adopting Scrum and eventually understanding what it means to be Agile.

3.1 Lesson 1: You can't get to Done (or Scrum) Overnight

This is very important to know when first starting out: actually implementing Scrum the way it is meant to be run can take months or years depending on the changes needed in the organization and becoming Agile is a never ending journey. This means two things:

1) If you really want to do it right, you need to be in it for the long haul.
2) Don't beat yourself up because you can't achieve exactly what Scrum tells you to do in the first few sprints.

Had someone told me this, I would have saved myself years of frustration as to why we couldn't implement this "simple" process. I believed the books and consultants that talked about this simple, light weight process which led me to think that it must be easy, right? WRONG. Scrum is only a light weight process because it relies on the fact that you are living in an Agile organization. That means that first and foremost you have an Agile culture that is the real glue, checks and balance system, driver, quality controller, continuous improver, and conscience of your product development. When you have a culture and a system that enables all of this, you can get away with having a super simple process like Scrum to manage your complex software development projects. I have seen scores of teams start their Agile journey with only Scrum thinking this will magically transform them into an Agile team. Scrum provides you with some visibility into what's truly wrong with your team and organization, but it doesn't tell you what the root cause is, how to fix it, or especially how to deal with the fact that a lot of organizations don't want to admit they're wrong or invest in making the big changes that will fix the core problems.

As an SQE, I implore you to start with the Agile values and repeat them over and over. Internalize them. Make everyone in the group internalize them. These will help guide you and enable the right decisions as you start to implement Agile methodologies like Scrum. Scrum and other Agile methodologies are very adaptable and can be tailored easily. It's critical that you have the Agile values and principles act as your compass to ensure you are always heading towards your ultimate goal. At the end of the day, this is all about business results and fundamentally providing value. While it may take a while to get there, focus on this will reap incredible rewards for your organization. Just make sure everyone understands it won't happen overnight. It's clear why your superiors must understand this (to prevent them from yanking the chain too quickly), but sometimes it's forgotten that not being able to achieve goals quickly can be demoralizing to teams. Help your team understand that this is a journey and celebrate each step you take towards being Agile or even getting the Scrum process right. This is your first sprint where you actually got a user story implemented, unit level tested, functionally tested, tested with the rest of the system AND it has zero critical defects? That's huge! Congratulate yourselves! Now figure out how you get more done next sprint and keep continuously improving. This is not to say that you shouldn't question why you're not actually doing Scrum. The team should understand why there are gaps and what they are going to do in order to fix them.

Eventually, when you get to a more Agile state and the team is really performing, you will see great execution and high quality every sprint. Our teams that were kept stable started executing like clockwork after four or five sprints. However, the rest of the organization was still in crisis fighting mode so these high performing teams weren't always recognized for the amazing accomplishments they were making each sprint. Angela Druckman, in her *Agile Transformation Strategy* white paper, notes that not recognizing these sprint over sprint efforts is a common failure in Agile organizations. As an SQE, you can help to highlight the great quality achievements being made each and every sprint on your teams.

3.2 Lesson 2: Champion Discipline

As mentioned above, when joining the Agile team I started as a ScrumMaster. I highly recommend SQEs joining this role if their team is moving to Agile. There are many similarities between the role, specifically the responsibility of championing discipline, adhering to core values

and principles, facilitating, and influencing without direct authority. The need to be an advocate for high quality software, discipline in development, and continuous improvement is critical in an Agile team especially when first starting out. Adopting Agile is a significant shift for the team and they need a strong rallier to push through the difficult parts. It is all too easy to fall back to what is comfortable (validating in the next sprint, individuals owning whole sections of the code on their own, piling up defects, etc.). We also know that we can't get to Scrum (done) overnight. This is where the SQE expertise can be applied. Knowing that we weren't doing full testing (scalability, performance, stress, etc.) every sprint, there was a false sense that we were really done done with stories. Someday, we may be able to perform full testing in a sprint – I've known teams that have accomplished this. But, in the meantime, as an SQE you can help by making the true state of the software transparent in other ways. Not only did I help the team define strict done criteria for each of their stories, I also defined sprint criteria that would help show all the testing that was done (and wasn't) in that sprint to stakeholders. That way, if the risk of not performing certain testing concerned the stakeholders, they could call it out and the team could incorporate it in the next sprint. Let's be clear, there was full functional and regression done every sprint. As an SQE, you have to help the team balance out what is cutting corners and what simply doesn't have any ROI to do every sprint. As much as possible, the amount not done each sprint should be a very low percentage of overall validation and be decreasing over time.

3.3 Lesson 3: Let Go – Resistance is Futile

Change is fundamentally difficult for humans and transformational change (such as moving from a traditional software environment to an Agile one) is even more difficult. The important part is to recognize that you will be undergoing a transformational change and learn some of the techniques for working through this change. Note that these types of changes can take years. For one of the small engineering teams I worked on, it took about four years to get to a high performing state. You'll gain a lot of benefit even early on, however. For SQEs, use this paper and start to understand how it will benefit your quality mission. I learned that everything I wanted I now had: a way to continuously improve, have a clear idea of the software quality, and a team that was carrying the quality torch with me.

During this transformation, it will be easy for the team to not follow all of the rules. For example, when I walked into the first Agile team I was on, they were validating their user stories in the following sprint from when they were developed. Obviously, this is not Scrum. However, at the time the team had accepted it and claimed doing both development and validation in the same sprint was impossible. It wasn't impossible however trying to accomplish this using the same development and validation approach we had used in the past *was* impossible. We had to change the way we worked and how we approached product development. That meant as a team we had to build up new skills and learn new techniques to accomplish this much higher state of software quality.

3.4 Lesson 4: Meet the Team

As Agile proclaims, individuals and interactions are more highly valued than tools and processes. The people doing product development are the most important part of getting high quality software out the door to customers. When moving to Agile, I've seen many teams try to squeeze people into roles where they lack the skill or ability to deliver on the responsibilities of the role. Everyone should take these roles and responsibilities very seriously because if anyone is half delivering on a role, you will not achieve your goals. Here is a description of the typical roles in an Agile team following Scrum and the insight I've gained as to what they really need to be.

Development Team: This is a team of self-organizing, highly disciplined professionals. They support and challenge each other, they are highly focused on getting things done correctly, and they are constantly monitoring the health of their team. At the end of the day, the team shares

responsibility for delivery of the product and there is no longer an excuse that any one person failed to deliver. Some examples of when you start to see this are the team swarming on stories, team members picking up tasks of others when they need help, and challenging each other to do things right from the beginning rather than banking on re-designing or fixing issues later.

Product Owner (a.k.a "Customer Whisperer"): This is one of the roles I see most abused in Scrum. Often a project or program manager is thrown into this position despite it requiring different skills. In Scrum, the team is managing their own work and executing, there no longer is a need to have someone tracking a schedule. However, there is a huge need for the Product Owner to balance out a large set of stakeholders needs, understand ROI, and see beyond what people are asking for directly. I call this the customer whisperer role because it simply isn't enough anymore to just take what customers say and write it down into a user story. The Product Owner has to look and see what the customer is ultimately trying to achieve and communicate this to the team. Why? Because as humans we aren't that great at saying what we really need and we typically phrase things in terms of what we know. Sure, I want a portable CD player that doesn't skip. However, what I ultimately want and am unable to voice is that I want a way to listen to music anywhere at any time without having to do any work. By communicating needs in this way to the team, the Product Owner allows the team of really smart engineers to innovate and create incredible solutions that delight customers (such as an MP3 player in this example).

ScrumMaster (a.k.a "Zen Master"): This is a very difficult role to become proficient in and in Scrum it is a critical role to get right. Teams may scramble around and get to a decent state without having a good ScrumMaster but they won't get to great. The ScrumMaster has to live and breathe the values and principles Agile espouses and be a leader for the team in this area. When teams are transitioning to Agile and everything is changing and many want to revert to a previous comfortable state, the ScrumMaster is the rock pushing the team forward. This person has to understand how to push when needed and how to stand back and let the team make mistakes in order to learn and ultimately become self-organizing. The most advanced teams I've seen put some of their most senior leaders in these positions because it ultimately does warrant this level of capability. Additionally, the ScrumMaster has to be able to continuously encourage the team to improve and get better. They are a zero-gravity thinker and must be able to step back and be an objective voice to the team.

Validation Engineer (a.k.a. "Developer in Disguise"): One of the biggest shifts for our team was realizing how much we could improve by having validation be competent in software development. If you're trying to complete stories, have them fully validated, and have low defects within a 2-3 week period all while keeping a sustainable pace, you no longer have time for validation to write out test cases and perform manual testing. You have to trim out activities that don't actually provide value. The validation team learned that they could write automated tests, comment in-line and auto-generate documentation of the test case while building up a regression suite that could be run on a nightly basis. This required them to understand how to write automated tests, how to work with the developers to design the software in a way that could be automatable, and even start to incorporate software development techniques like code reviews on their own tests.

Developer (a.k.a. "Code Artist): As we started to remove all the unnecessary activities that developers have to deal with in a typical product development environment, we were able to allow the team to spend more time on writing really good software. I understand this is a novel concept but in the typical development I had seen there was simply too much thrash, unrealistic schedules, and lack of discipline to have time to think about writing really great code. As we started to remove impediments from the team and create an Agile culture, developers were able to spend time on design and developing a flexible architecture that would support changes late in the project. They also wrote code in a way that was clean and understandable because anyone on the team should be able to work on any part of the code when needed. Furthermore, they focused on designing in a way that would limit technical debt and enable automated testing. The Agile team is not the disparate group of programmers slapping together code and throwing it over

the wall. This was a team focused on creating value on a continuous basis and writing software that would enable them to be agile regardless of changing requirements at any stage in the life cycle.

Manager (a.k.a. "Servant Leader): With the team being self-organizing and the ScrumMaster focused on developing the team and creating a culture of continuous improvement, where does the manager fit into this picture? The role of the manager is different on an Agile team and may or may not be necessary depending on the organization. What we found is that the manager is now a career coach focused on helping people to develop. Other than that, most of their time is spent on removing impediments for the team. They clear a path to let the team execute. Do they need to know about the day to day activities of the team? No. Fortunately, this allows managers time to focus on the bigger changes need in the organization to make their team successful.

To close this discussion of how roles and responsibilities change on an Agile team, it's important to note some of the roles that are often no longer needed. We will need to say our goodbyes to the extra roles we've built over time to manage a process that doesn't fundamentally work. Goodbye to the Project Managers, the "Leads" on teams, and even to the Architect. The responsibilities of these roles are absorbed in the roles mentioned above. The team manages their own schedule and execution and the development team is now a group of leaders and great software designers.

3.5 Lesson 5: Let's get Technical

When starting Scrum, we had this false impression that by just trying to adhere to Agile and follow the Scrum process we would magically be able to achieve Scrum. Don't let anyone tell you this! We struggled for a long time with Scrum until we had people join our group who had experience in Extreme Programming (XP) and were able to bring XP techniques to the team. XP practices give the development team a way to get to clean code and increase discipline in reducing or ultimately avoiding technical debt. To repeat again, Scrum is just a project management process and you need to develop an entire system in order to deliver high quality software. That means the development team has to employ Agile technical practices in order to deliver to this goal.

Conclusion

The road to Agile can be bumpy and is far from a straight path. It requires critical thinking every step of the way, closely monitoring key metrics (delivery of working software, team health, technical debt, etc.) to understand where impediments lie and how to remove them, and the maturity to let go of old, comfortable ways of working in favor of transformative change that will lead to creation of a system that delivers the highest business value possible. The rewards and lessons learned in this paper are a few of the key things I wish someone would have made brutally clear to my team as we started our Agile journey. There are many other complexities and activities that are part of being Agile and even achieving a Scrum process so this is by no means a complete set. For those that still have the gumption to adopt Agile, I wish you all the best and hope these lessons help! For us, it has been an extremely positive transformation to the point that no one on the team can imagine developing software in any other environment. The day you see that you can count total software defects on two hands, the whole group is focused on delivering customer value, schedule slips are a thing of the past, and the team is continually engaged and having fun you'll look back and understand the power behind creating a whole quality system.

References

Druckman, Angela. "Agile Transformation Strategy." Collabnet, Inc., 2011.

Beck, Kent and others. "Agile Manifesto," http://agilemanifesto.org. 2001.

Stacey, Ralph. D. *Strategic Management and Organisational Dynamics: The Challenge of Complexity*. Prentice Hall, 2011.

Distributed Agile Teams are FASTER THAN A SPEEDING BULLET!

Aaron Akzin
aakzin@webmd.net

Abstract

Are distributed teams more powerful than a locomotive? Maybe.

Are distributed teams able to leap buildings in a single bound? Well…..

Can Agile development work with a distributed team? Given that the Agile Manifesto calls for "Individuals and interactions over processes and tools" does having remote workers doom a team to reverting to non-Agile methodologies? It's a business reality that staff won't necessarily be co-located which can make adopting Agile more challenging due to a lack of face-to-face interaction and often high latency communication channels.

This potential hindrance can be turned into a strategic advantage when applied to an Agile team. How do you leverage those resources in staggered time zones to extend the work day, productivity and improve development cycle time? What tools and techniques help remote workers stay connected to their team? How can cultural and geographic diversity provide a uniquely satisfying work experience for all team members? This paper provides a case study of how a team effectively navigated the challenges of a distributed Agile team.

Biography

Aaron Akzin is a Senior Quality Assurance Analyst at WebMD with over 13 years' experience in technology primarily in a testing engineer role and mostly in health care organizations.

Sudha Sudunagunta is a Senior Quality Assurance Analyst at WebMD with over a decade of experience in various technology organizations.

1 Introduction

Many organizations have made the successful transition from Waterfall or no methodology to Agile development and life cycles. It has paid dividends for many while proven to be a struggle with limited success for others. There are 12 principles of Agile development that are outlined on the official website (http://agilemanifesto.org/principles.html).

One of these 12 principles is the following:
"The most efficient and effective method of conveying information to and within a development team is face-to-face conversation."

We all recognize this as being an ideal scenario. Unfortunately, there is very little guarantee that proximity will be our work environment with our co-workers. In fact, it's quite common at large organizations that resources and team mates are geographically diverse and often are in different time zones. Quite frequently they even have language barriers (as well as cultural) differences to surmount. In addition, talented staff may move for personal reasons to other locations. Often, organizations wish to retain their knowledge in-house and take steps to accommodate geographic diversity while still maintaining their Agile techniques and not revert to Waterfall or worse yet, no methodology at all.

Despite this principle there are advantages to a geographically diverse work staff. First, time zones can work to your advantage as an organization if leveraged properly. It is quite possible to dramatically increase your productivity and increase the number of cycles you can achieve in a 24 hour period versus a normal business day. At WebMD Health Services, we have two off-shore partners that we partner with that help us achieve these aspirations. It has been our experience that the distributed nature of our staff has often increased our throughput and velocity and is the rationale behind the title of this paper.

One off-shore partner is primarily dedicated to testing and located in India. The time zones are such that their night is our day and vice versa. There is a small amount of overlap due to the efforts of the off-shore partner slightly shifting their schedule to permit some degree of simultaneous communication. During this window both staffs are available to share results from testing as well as inquire about uncertainties or even used as opportunities for learning and dissemination of knowledge.

Personally, I had an opportunity to benefit from this time zone difference many times in the role I currently hold. One of our products required a database build every 2 weeks with updates to a provider directory used by consumers. Our ambitions were to move to a 1 week cycle and still feel confident on the integrity of our build without sacrificing testing cycles. There wasn't a tremendous opportunity for automation initially. Our partners in India didn't have much knowledge of this product line. Time was allocated to train the staff over a period of a few days at the beginning of our day and conclusion of theirs. Then, thorough test plans with explicit steps were authored so remote staff was able to execute precisely what was needed. In a very short period of time, what evolved was a complete cycle of work. The database build would be completed in the late evening of our business day and be ready for verification at the beginning of their day. Testing results and a corresponding matrix were recorded at the onset in an Excel spreadsheet. Eventually this evolved into more real time tools such as Google docs/spreadsheets and wiki pages which gave more opportunity for version control and multi-user capabilities.

An additional off-shore partner is tightly embedded into our entire Agile strategy. This partner is based in Argentina and the amount of hours where there is overlap between them and the United States employees is much greater and increased by a willingness on the part of that partner to slightly shift schedules, as well. Engineers in Argentina are both Quality Assurance as well as Developers. They are fully functioning engineers as productive and counted upon as a valuable resource as those based in the United States. They are interspersed in all our small teams and participate in our daily stand-ups. In addition, they are involved with committee meetings that help us to achieve our corporate goals. I have had the opportunity to meet many of the engineers that are based in Argentina at the onset of working with them. I will provide more details on why this opportunity to meet in person is ideal and in many ways necessary towards achieving a bond with your teammates.

Lastly, there are members of the WebMD Health Services staff that are engineers that work remotely and only take periodic and intermittent trips to our division office in Portland, OR. For example, our Chief Architect resides in Northern California and makes very frequent trips to the office. Another member of the Technology department resides in Nova Scotia, Canada and while his trips are more infrequent given the distance and expense he still makes the pilgrimage to the home base on a scheduled basis.

In fact, I am a remote employee that resides in Orange County, California and approximately on a quarterly basis makes the trip to Portland, OR. Candidly, this geographic status I enjoy drove me to author this paper and become an advocate for how it can fit into and even add value to the Agile model your organization adopts. It happened through an unusual set of circumstances that I didn't anticipate, however, deeply love and cherish this benefit. I feel connected to my co-workers that I work with even though I don't sit near them each day. I see them; I hear them and communicate with them frequently. I don't consider myself isolated in any way. On occasion, I jest about the "Island of QA in Orange County" staffed by myself and "Wilson my volleyball companion", it isn't accurate, whatsoever. Part of this is through nominal efforts on the part of WebMD Health Services to put in place simple tools to allow for better communication and invest in low cost towards travel and technology. It is my personal belief that many people would feel greater satisfaction with their job and work life balance if they also enjoyed the ability to work without a commute and the associated time and cost with that normal effort. Flexibility retains employees and talent as much and if not more than increasing salary. It is certainly worth investigating and formulating a strategy on how to permit distributed staff and still maintain an agile work flow and life cycle.

2 Strategies for Communication

The most important strategy for any team, whether they be co-located or geographically diverse is frequent and easy communication. Below are a number of strategies to and techniques for communicating and ordered in the way that has the greatest probability to achieve a bond among team mates. The easiest route to failure is a team that doesn't enjoy working with each other and lack of familiarity leads to distrust.

2.1 Meeting in Person

Despite the plethora of technologies that exist to communicate with others via email, IM, VOIP and other techniques as noted above there is still a strong incentive to congregate as a team at the onset of working together and periodically after that to maintain a sense of community. I work remotely for WebMD Health Services and my trips to the office to be around my team generally occur on a quarterly basis. On these trips, there is always some effort to have some sort of team building event. I'm going to provide a few examples below of some of the events and activities that have occurred in the past.

One of our more physical events was an off-site activity at a place called Tree-2-Tree (http://www.treetotreeadventurepark.com/). There were two portions to the team building event. First, the team participated in a number of activities that are pretty standard. The first half of the day involved working with one another to figure out a somewhat daunting task or relying upon team mates in order to achieve a goal that the staff of Tree-2-Tree gave to you. For a person afraid of heights such as myself it wasn't an entirely exhilarating day (on the contrary, quite frightening at moments), yet it was still a wonderful experience to get off-site and bond with your co-workers in some sort of physical event.

Music has an extraordinary effect on us as human beings. In a work context, it can bond us with each other. In the course of my working with my Distributed Agile team I have sung Bohemian Rhapsody in Karaoke with Argentinians. We have all played Rock Band together. We have memories of each other and enjoying each other's company in a social setting. If you don't have that as the foundation of your team whether they are co-located or distributed it is much harder to achieve a cohesive and well-functioning team.

The most critical part of congregating teams is to have a memory that they can take with them when they return to their various geographical locations. The shared memories lend themselves to cohesion and friendship and one that management should really budget for when bringing in new players or instantiating a new group.

2.2 Social Networking

One of the most important aspects of keeping a cohesive team that enjoys interacting with one another is socialization. At WebMD Health Services, our ambition with one of our off-shore partners from the onset was tightly embedding their engineers into our teams. As noted earlier, they participate in the daily stand-ups as does everyone.

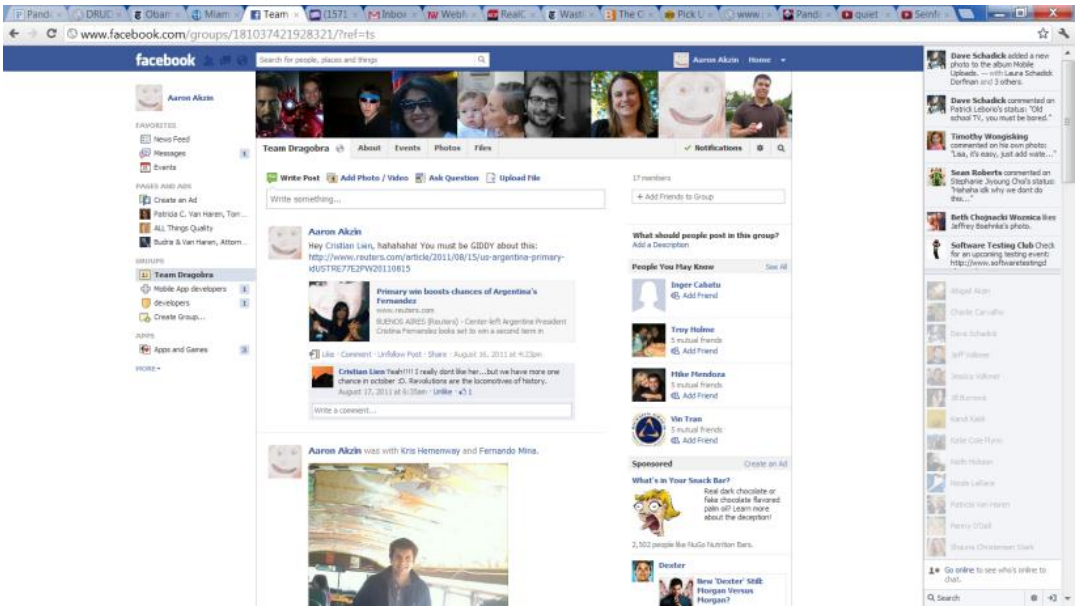
Facebook is an example of a site that presents a wonderful opportunity to personalize yourself to your co-workers in a way where you might not otherwise get that opportunity. Social networking sites like this or Twitter and Tumblr are so common to people regardless of what country they live in that there is an eagerness to utilize this modern form of becoming a “pen pal” to correspond or share with far away people.

In early 2011 WebMD Health Services adjusted the team allocation as we frequently do in our constantly evolving process to improve our organization. Previously, the engineering staff based in Argentina that is so embedded in our Agile methodology was interspersed in each team. At this point in our evolution of using these resources, it wasn't the right moment to structure ourselves in that way. Unfortunately, it made teams less productive since each team needed to spend additional time training and acclimating remote resources to the complexity of our work. Also there are restrictions that are specific to our business line where we are not able to have off-shore resources working on all aspects of what we are doing.

As an alternative to this original structure we congregated all the Argentinian staff onto a single team and provided a ratio whereby there was at least one US based resource for each foreign engineer. The rationale was to provide more dedicated and available resources to be readily available to foreign based staff while they acquired a pocket and library of knowledge in that part of the world. After that ambition had been achieved these resources would be redeployed and interspersed to the other teams as had existed before. It was just recognized the inevitable decrease in productivity as you train new staff to learn to do complex work.

I was recruited for this team and assigned an Argentinian engineer that I was to partner with and reach out frequently to throughout the day. After communicating with him in email, IM and Skype I realized what we were missing in our attempts at integration of this off-shore staffing. There wasn’t a socialization aspect that had really been recognized much less any attempts to bridge this chasm. There is an inevitable aspect of socialization that is necessary to achieve a cohesive team that works with each other and very importantly, enjoys it.

At that time I was quite interested in Facebook for a myriad of reasons. First, I was fascinated by the technology and the number of simultaneous world-wide users all on this platform at every moment of the day. Second, it was the new “thing” in the world of technology and was monopolizing all the buzz, interest, career opportunities and investment for some period of time. Third, I found it a wonderful outlet for socialization myself while I was working throughout the day and off-hours when I wanted to reach out to my peers. I realized it was a readily available and free opportunity to break down some gaps that had occurred in bonding team mates. Thus I started a Team Group and invited all members of the group into this Facebook “party” and requested they all take photos of their respective offices so we could have some vision of the work space that each party enjoyed.



At that moment I was the only employee of the branch office Orange County and took photos of my townhouse. My son was photographed as the “facilities manager and maintenance supervisor” while my daughter was crowned with the title of “office manager and receptionist.” A photo was taken of my “executive” view and washroom. It was a Saturday afternoon attempt at humor that was well received by my colleagues in Argentina that were eager to reach out and bond with their American based team mates. In return they saw these photos and on Monday returned the favor by photographing their own office space which included their Yoda mosaic mural, bean bag chairs and Wii console area for ad hoc tennis tournaments. Shortly after this was done we had the opportunity to meet in Portland for a week and the sparks of friendship had already been achieved.

What was significant about this development is we had an opportunity to see the environment in which our peers worked lending a greater understanding as to how they worked as we did and where it differed. Also, it provided an opportunity for conversation in daily stand-ups to inquire and follow-up about some photo or event we might have seen our team mates share on their profile or within our Team Group.

2.3 VOIP

Voice Over Internet Protocol is an acronym that in many ways has changed our lives and dramatically lowered the cost of operating your business, particularly when resources are international. If you haven’t experienced this yet, you are missing out on a wonderful application of the internet, calling others through your computer instead of by phone at very low or no cost. Skype is generally our method of speaking with staff that is international. However, Microsoft Lync is often used in this same capacity as it has been adopted as our corporate solution.

In addition to the lower cost of VOIP for businesses, I have recognized a higher quality in terms of sound and audio in hearing. This becomes quite important when you are calling into a meeting throughout the day. At WebMD Health Services, many of the conference rooms have a ceiling that is quite high or even open given the office space that is occupied. I have found that in meetings where VOIP is used instead of a traditional phone, I don’t find myself struggling to hear as much of what is being said. Despite this increased quality and clarity, there are still "quiet talkers" (http://www.youtube.com/watch?v=1_3UmqEAh8c) where even if you were on the other side of the room there would be some difficulty hearing all speakers. This can be particularly frustrating as a remote employee and on occasion I have communicated with a trusted team mate that was present at the

meeting via IM after the meeting has concluded to reiterate certain components of what was communicated that I may have missed.

An important investment organizations need to undertake and provide for employees is a quality headset that facilities speaking. I purchased a wonderful Logitech headset that I use to listen to music on Pandora.com throughout the day as well as benefitting from it for collaborating with my peers and participating in meetings. The cost of this headset was nominal, only $60 and among our team of 10 can be achieved quite easily, financially.

2.4 Screen Sharing Tools

As time has gone on, there are more and more methods to share your screen and work interactively with a remote resource. When I first started in technology in the mid-90s this was a luxury and almost always facilitated better communication while mitigating a lack of comprehension of what one person was seeing while the other wasn't. Back then, we would often settle for screen shots and annotations with recreation steps. However, that was often lacking an important piece of information that was necessary in order to properly capture the sequence of steps required to recreate a problem or disseminate knowledge.

Many tools we use daily have screen sharing functionality built into the product. Skype and Microsoft Lync are two wonderful examples where you are already engaged in an IM conversation or even VOIP and taking the additional step to share your screen is free and easy. The screen sharing technology has become so ubiquitous that popular sites such as join.me offer this exclusively for free with the hope of upselling you to richer functionality after they have hooked you on this benefit.

2.5 Streaming Video and Webcam

The Agile team I work with every day does our daily stand-up with Skype for VOIP and GotoMeeting for video. It is useful and humanizing to have the streaming video as part of this conversation. An important aspect of maintaining good relations for the team is a level of personalization that is more easily accomplished when all resources are in the same geographic location. Although, I think all have had an experience where we may wish that some resources weren't so close....

There is not necessarily an added business value by being able to see your co-workers as part of your daily interactions with them. As a remote resource I am able to function perfectly well without seeing my co-workers for months at a time. Still, as a human being I value and appreciate being able to see them each day. Since we are all human beings, of course, then there is a business value as it eases my comfort and familiarity with my distant co-workers. For example, seeing the clothing your co-workers wear and how it might differ from you gives you the opportunity to appreciate the respective locations you may all be in and fully comprehend the heterogeneous nature of your peers. I reside in Southern California and quite frankly forego socks for many months of the year and certainly long sleeves. In comparison, the majority of my team mates reside in Portland, OR where the weather is much wetter than it is for me. Often we will tease one another about our climates where I may not see rain for months at a time and they go almost daily with some sort of moisture. The disparity in our weather becomes even more apparent when it involves the engineering staff in Argentina that is below the equator so their seasons are the reverse of ours.

Every morning I see a streaming video such as this (GotoMeeting screen shot below) where each team member does their daily stand-up. There are four engineers in Argentina, 3 in an office together and another that works in her home hundreds of miles away. There is myself based in Orange County, the branch office of Team Cobra as I joke to myself and my co-workers. Then the remainder of the staff on my team is located in Portland, OR where they gather in a room equipped with a webcam, microphone and computer where we can provide our update as well as scrutinize our electronic Kanban together.



2.6 Instant Messaging

Instant Messaging was initially a social mechanism for people to interact rapidly entirely in the context of our personal lives. There were no ambitions that it would be applied to business avenues. Instead it was adopted by many who used it in their personal lives and desired the same rapidness in their work that they were able to achieve in their personal lives. In fact as time has passed, corporations have purchased rather expensive corporate enterprise solutions to permit and facilitate Instant Messaging among employees. For those of us that recall how hostile many employers were to the concept of Instant Messaging it is extremely ironic.

On a given day I communicate with my co-workers through many IM technologies. I may vacillate between Microsoft Lync, Skype IM, GTalk, AIM, Windows Live or Yahoo Messenger. The only problem becomes when staff uses different IM tools, although solutions exist that bridge these Instant Messaging clients such as Pidgin. I would say IM is my primary method of communication followed closely by email. In addition to the obvious advantages of speed in IM, there is a certain amount of informality that lends itself to that venue and can help achieve familiarity and friendliness among team mates that don't get the opportunity to visit each other in person on a daily basis.

2.7 Email

Electronic mail is a method of communication that is so ubiquitous now it may hardly seem worth noting as a fabulous forum for communication. The value of having communication you can easily store and retrieve and add additional resources for input and collaboration is well known. However, in the context of Distributed Agile teams, often you have non-native English speakers communicating with the team. These resources based in non-English speaking countries have often learned to write and have an easier time communicating more clearly writing versus orally.

Slang and localisms are not often taught, comprehended or familiar to foreign based staff. An amusing anecdote I have pertaining to this subject is a conversation that occurred on my team a year ago. My first name, Aaron, has generally been uncommon at many organizations I have worked at over my career. However, at my current employer, WebMD Health Services, there are a number of Aaron's within the technology department. In fact, there were two Aaron's on my team, myself and another. This became a somewhat confusing experience during our daily stand-up since often when you differentiate between individuals you might make eye contact with the resource you wish to provide an update. In our case this wasn't too practical given our different locations and at that time we were exclusively using audio without a video component for daily stand-ups. In jest and to alleviate confusion I stated I should be referred to as "Thing 1" and my co-worker as "Thing 2." Staff based in the United States immediately understood the reference to Dr. Seuss and it was quickly adopted in the daily stand-ups thereafter as well as in an occasional email. After a week or two subsequent to this being adopted, staff in Argentina inquired on the meaning of this reference. It was asked in an IM to me "what is this Thing 1?" I was quite amused and found the following YouTube video (http://www.youtube.com/watch?v=r0F8rYqvmEw) and provided some additional explanation so that it made sense to those team members.

It was an important lesson for me. How you should consider even small references of culture and what effect they may have in terms differentiating culture and most people (particularly technology professionals) are genuinely interested in learning about other cultures.

3 Summary

In conclusion, most people recognize we won't all be working in the same location. We may want that…or not. Regardless of our desires it is impractical to believe we will all be co-located. So how do we bridge the gap of geography, time zones and culture? We do it by achieving a bond and attempting to make a connection on a personal level prior to working as peers. It is easier for people to work with one another after you have positive feelings towards another individual. I know over the course of my career I have always enjoyed working with people I like that may not be as gifted technically or experienced if it is accompanied with a personality that I find pleasant that there are pleasant interactions and hopefully humor and shared interests.

I consider myself quite fortunate that I am not burdened on a daily basis with a commute. It is good for the environment, gives me more time in my day and permits me to reside in the location I wish in close proximity to my extended family. That by itself would be a gift from my employer to me while also being considerate enough to pay me a salary. In addition, I am afforded the opportunity to work with resources 1000 miles away in the Northwest of the United Sates, on the other side of the Hemisphere in India and the on the opposite side of the equator from me in Argentina. We have entered a new phase of technology where our opportunity to interact with different cultures and gifted engineers raised in another society with their own respective culture is one that can enrich my knowledge as a global citizen.

Most importantly in order to innovate we benefit from exposure to technical problems that must be circumvented. One of those technical problems is in fact social. Those of us that gravitate to engineering aren't adept or known for our social grace and ability to adapt to human emotions. On the contrary, we find ourselves ignoring a natural instinct we all have that fears or at least withdraws from change. Fear of change is fear of differences and Distributed Agile teams should embrace the opportunity you are afforded. How often do you get to have your morning coffee and talk about your plans for your workday while that other person on the screen is sipping a mate (http://en.wikipedia.org/wiki/Mate_(beverage)) ?

Lightweight extensible provisioning system to stage heterogeneous multi-machine test environments

Vladimir Averkin vladimia@microsoft.com
Muhammad Usman Janjua mjanjua@microsoft.com
Sunil Kutty sunilku@microsoft.com

Abstract

A modern app is a connected app – distributed, spanning multiple machines. How do we simulate the varied execution environments for such an app? How to stage an app that requires multiple machines working together in a finely tuned, elaborately configured ensemble? How to deploy complex topologies repeatedly and reliably, support detailed configuration, and facilitate real-world testing simulation in a controlled lab environment? How do we accomplish all these with a self-contained module, which is independent of test execution details and highly pluggable with varying test execution infrastructure?

We attempt to provide a solution based on virtualization and lightweight script-driven automation. Employing technologies of Hyper-V, virtualized storage, Windows PowerShell and Windows Management Instrumentation, we deliver a solution to automate deployment of complex multi-forest multi-tiered test topologies, with elaborate machine-level and cross-machine configuration, and a fully consumable topology definition. Hyper-V facilitates fault-injection and destructive testing on topology machines which are easily restored to states saved earlier.

With the imminent evolution to centralized test-labs in distant cloud-farms, we are bound to face the challenge of decreasing physical access to machines and less control and flexibility of custom-created test environments. A lightweight, agile, scalable and extensible solution available in-house, capable of creating highly customized test environments in a rapid and repeatable pattern is bound to prove very useful. An implemented concept of such a solution is what we are going to present here.

Biography

Vladimir Averkin is an SDET II at Microsoft, currently working on testing Server Manager's role and feature deployment for the next version of Windows Server operating system. He joined Microsoft in 2007 and was a key contributor in the development of cross-machine test infrastructure to test different remoting scenarios for Windows PowerShell, WS-Management and WMI.

Muhammad Usman Janjua is an SDET II in Microsoft Windows Server Division. He did his masters in computing at Johns Hopkins University. Muhammad Usman joined Microsoft's Windows Manageability Team in 2007 and has since owned server manager remote management and deployment features.

Sunil Kutty is a Senior SDET Lead in Microsoft Windows Server Division. He holds a bachelor's degree in Electronics Engineering and an MBA from University of Washington. Sunil joined Microsoft's Windows Manageability Team in 2000, and has been working on the entire management stack ranging from WMI, WS-Management and Windows PowerShell to Windows Server Manager.

1 Introduction

Today's application environment spans multiple machines working together as a finely tuned, elaborately configured mechanism. To test such a distributed application has always been a challenge, where initial deployment and configuration have frequently been more complex than the testing itself. How can we deploy those machines, configure them and orchestrate testing that spans machine boundaries? How can we do this and yet have lean and transparent test infrastructure with minimum impact on the execution environment? How can we avoid using expensive 3rd party tools and keep our testing as close to real-world customer scenarios as possible?

The answer to this is to use virtualization and script-driven automation, two technologies that are quickly becoming more prevalent in today's IT landscape. With the technologies that are part of an operating system, such as Hyper-V, Virtualized Storage, Windows PowerShell, and WMI, it is finally possible to automate increasingly complex deployment scenarios. With virtualization, it is also possible to run potentially destructive tests that can put machines to a bad or unreliable state and restore the machines back to a stable state after the verification.

The biggest problem with the existing solutions is that none of them can automate the deployment process end to end. At some point there is always a series of manual configuration steps required. Second important problem is the use of special tools to control the deployment. The commonly used approach is to employ some agent programs to execute a set of instructions sent remotely from a controller machine. That makes the test topology unclean, introducing things that are never going to be part of production environment.

Our approach is to use a set of in-box operating system capabilities. All those capabilities are well-known and documented. In addition, we use several public tools that are built on top of those capabilities.

The key technology that has brought a revolution in machine provisioning is virtualized storage or VHD. The VHD is the basis for virtual machine creation and starting from Windows 7, booting physical machines from VHD is also supported. Coupled with an old and well-known multi-boot technology, this simplifies provisioning of physical machines from the same virtualized media.

Another wonderful technology is Windows PowerShell. This automation superglue has an amazing ability to bind together various software technologies created by Microsoft during the past 15 years, from COM and WMI to .NET and Hyper-V.

The deployment is not managed from one central location. Instead, VHDs are "primed" with a unattend.xml and a set of supporting scripts and other files to do deployment by themselves. Once the OS deployment starts on provisioned machine, it goes entirely on its own. No network connectivity is required at this point. The actions are daisy-chained in such a way that no human intervention is required and once they finish, the machine is fully deployed. During the deployment, the machine can go over one or more reboots. The ability to do configuration that requires multiple reboots is especially valuable for deployment of domain controllers, which is necessary when creating multi-domain topologies.

This approach is not only a more natural way of deploying test environments, it can also benefit its users as an educational tool. All the code is script-based, which makes it easy to distribute, and its design and implementation details can be clearly seen from the code. This makes it useful as a transparent showcase demonstrating how various technologies work and interact with each other.

2 How it works

2.1 Basic outline of the provisioning system (aka Machine Factory)

Provisioning system is implemented as a Windows PowerShell module, called MachineFactory. To deploy the module to your machine you simply need to copy the folder containing the module and supporting files to the machine which you want to use to launch topology deployments.

Getting started with MachineFactory module is really easy and, in general, boils down to a handful of commands executed from Windows PowerShell console:

```
Set-Location C:\MachineFactory
Import-Module .\MachineFactory.psd1

# Check if the specific OS version exists in a particular build branch.
Get-BuildMedia -MediaType VHD -Platform Server -Branch branch_main Locale de-DE

# Execute WhatIf (simulated) deployment to make sure there are no problems with
# factory configuration and topology definition.
Install-Topology SimpleTopology.xml

# SimpleTopology.rdg file is generated at the end of deployment which
# can be used with Remote Desktop Connection Manager to connect to machines.
# Use this deployed topology for executing tests, running validations etc.

# Finally here is the command to recycle the topology and free up resources
Uninstall-Topology SimpleTopology.rdg
```

The module itself has about 12 commands, of which the knowledge of only 3 is required for regular usage. Those three commands are:

- Get-BuildMedia
  The command is used to check the availability of the media for a particular platform, branch, locale, etc.
- Install-Topology
  Takes a topology definition defined in XML file and deploys a corresponding set of machines.
- Uninstall-Topology
  The command discards machines that are no longer needed to free up resources.

All cmdlets are script-based as opposed to compiled C# for simplified maintenance

2.2 Initial configuration of the system

In the current version, system configuration is done via manual editing of the following configuration files:

Factory.xml – stores a list of virtual and physical hosts, minimum resource requirements for the host and encrypted logon to get access to remote shares using CredSSP authentication.
Media.xml – stores well-known locations of released versions of Windows
Tasks.xml – stores a set of deployment and configuration scripts, e.g., join machine to a domain, add user to administrators group, enable remote desktop, etc.

```
<?xml version="1.0" encoding="utf-8"?>
<FactorySettings>
    <Organization>MSP</Organization>
    <DefaultBranch>main</DefaultBranch>
    <MasterAccount>
        <UserName>TestDomain\FactoryAdmin</UserName>
        <Password>AQAAANCMnd8BFdERjHoAwE/Cl+</Password>
    </MasterAccount>
    <ResourceRequirements
        ReservedHostMemoryMB="4096"
        VMDiskSpaceGB="20"
        VhdDiskSpaceGB="40"
    />
    <VirtualHosts>
        <Server Name="VMHost01" VMFolder="D:\VM" VHDFolder="D:\VHD" />
        <Server Name="VMHost02" VMFolder="D:\VM" VHDFolder="D:\VHD" />
    </VirtualHosts>
    <PhysicalHosts>
        <Server Name="PSPN250311 VHDFolder="C:\VHD" />
    </PhysicalHosts>
    <VHDStoragePath>\\aapn1203981.microsoft.com\VHDCache</VHDStoragePath>
</FactorySettings>
```

Figure 2.1 *Sample Factory.xml file*

These files use proprietary XML format to store information, but the format is very straightforward and it is easy to modify the files once you have the examples. We plan to switch to PowerShell script-based configuration later on to improve discoverability of configuration data parameters.

Notice that machine factory stores the logon information. That becomes useful when we connect from computer that runs the factory scripts to remote hosts and try to copy media and other files from network shares. Accessing remote location from another remote location is known as a “second hop” scenario and requires enabling CredSSP authentication for local and remote computers and explicit credentials. To avoid requesting credentials each time we require access, we require them to be set only once and persist them as encrypted secure string, also encoded with Base-64 to store in the XML file.

You will need to run Set-FactoryMasterAccount cmdlet before the first deployment to get and save the logon information.

2.3 Simple deployment scenario

We will explain how provisioning works using a fairly simple deployment scenario. Imagine that we need to test how one computer can be managed remotely from another computer – a typical IT management scenario.

For this, we need to deploy two Windows Server 2008 R2 machines with the following specs – *RAM=1GB, HDD=60GB, Locale=en-US, SKU=ServerEnterprise* as shown on Figure 2.2. We also want to configure the machines to allow remote desktop access and join them to our test domain.

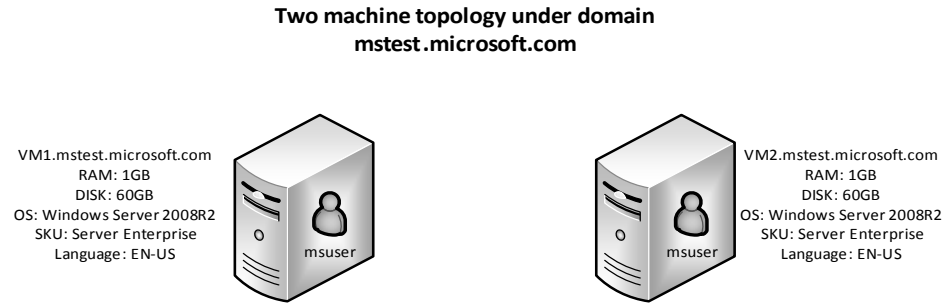
To deploy this configuration you need to run the following command:

```
Install-Topology SimpleTopology.xml
```

Before doing this, you can run this command with `-WhatIf` switch to check if the defined topology can be deployed successfully.

```
Install-Topology SimpleTopology.xml -WhatIf
```

WhatIf execution will go through all deployment steps in simulation mode without actually deploying machines. This is useful when you want to check if there are any problems with factory configuration, hosts connectivity, topology definition files are correctly written, etc.




```
<?xml version="1.0" encoding="utf-8"?>
<Topology Name="RemoteDeploymentTest">
    <Description>A topology of two servers for manual testing of Server Manager
deployment scenarios.
    </Description>

   <RunOnFirstLogon>
        <SetNetworkLocationPrivate/>
        <SetBackgroundInfo/>
        <DisableScreenSaver/>
        <EnablePing/>
        <EnableRemoteDesktop/>
        <JoinDomainWithRetry DomainName="mstest.microsoft.com"
UserName="mstest\msuser" Password="passw0rd"/>
        <AddUserToLocalAdministrators UserName="mstest\msuser"/>
       <SetAutologonAndReboot DomainName="mstest" UserName="msuser"
Password="passw0rd"/>
        <ShowFileExtensions/>
        <EnableQuickEdit/>
    </RunOnFirstLogon>

    <Group Name="mstest" UserName="mstest\msuser" Password="passw0rd"/>

    <Machine Name="Server2008R2" Count="2">
        <MemorySizeMB>1024</MemorySizeMB>
        <DiskSizeGB>60</DiskSizeGB>
        <OSArchitecture>amd64</OSArchitecture>
        <Platform>Windows Server 2008 R2</Platform>
        <SKU>ServerEnterprise</SKU>
        <OSBaseLanguage>en-us</OSBaseLanguage>
    </Machine>
```

Figure 2.2 *Simple two-machine topology and a topology definition file used to deploy it.*

2.4 Defining configuration tasks

As you can see in the SimpleTopology.xml file, there are some configuration tasks defined under `<RunOnFirstLogon>` element, e.g., `<SetNetworkLocationPrivate/>`, `<EnableRemoteDesktop/>`, etc. Where are those coming from? To understand this we need to look at the tasks.xml file part of which is given on the Figure 2.3.

You can see that the configuration task is actually a CMD script wrapped in XML and decorated with some metadata, notably, task name and description. So when we put task reference like `<EnableRemoteDesktop/>` into topology file, we actually instruct the deployment tool to inject this script into unattended execution chain on the related VHD file. Once the configuration task is referenced, the provisioning engine generates a **.cmd* file, corresponding to configuration task, e.g., in our case, `05-EnableRemoteDesktopTask.cmd`. It will also create a corresponding entry in the *unattend.xml* file, which is placed to the root of VHD drive:

```
<SynchronousCommand wcm:action="add">
       <CommandLine>%SystemDrive%\Deployment\05-EnableRemoteDesktopTask.cmd &gt;
       %SystemDrive%\Deployment\05-EnableRemoteDesktopTask.log
       2&lt;&amp;1</CommandLine>
       <Description>Enable remote desktop.</Description>
       <Order>5</Order>
</SynchronousCommand>
```

This somewhat enigmatic chain of characters under `<CommandLine>` makes sure the configuration script will be executed when the VM comes alive and will go through unattended execution phase. It also redirects all the console output generated by the script to a log file, which can be analyzed later, in case the deployment did not go as expected.

```
<?xml version="1.0" encoding="utf-8"?>
<Tasks>
    <Task Name="EnableRemoteDesktop">
        <Description>Enable remote desktop.</Description>
        <Execute>
            reg add "HKLM\SYSTEM\CurrentControlSet\Control\Terminal Server" /v
fDenyTSConnections /t REG_DWORD /d 0 /f
            reg add "hklm\SYSTEM\CurrentControlSet\Control\Terminal
Server\WinStations\RDP-Tcp" /v UserAuthentication /t REG_DWORD /d 0 /f
            netsh firewall set service type=remotedesktop mode=enable profile=all
        </Execute>
    </Task>
    <Task Name="SetNetworkLocationPrivate">
        <Description>Set network location type to private.</Description>
        <Copy Source="\\msdvault\msdvault\MSPPERF\vladimia\Tools\sleep.exe"
Destination="0:\Tools"/>
        <Copy
Source="\\msdvault\msdvault\MSPPERF\vladimia\Tools\SetNetworkLocationPrivate.exe"
Destination="0:\Tools"/>
        <Execute>
        <![CDATA[
             REM Wait for network to come online.
             %SystemDrive%\Tools\sleep.exe 30
             REM Log network state for diagnostics
             ipconfig
             %SystemDrive%\Tools\SetNetworkLocationPrivate.exe
             REM --- Do not show network location UI on logon.
             reg add "HKLM\SOFTWARE\Microsoft\Windows
             NT\CurrentVersion\NetworkList\NewNetworks" /v NetworkList /t
             REG_MULTI_SZ /d "" /f
        ]]>
    </Task>
    …
</Tasks>
```

Figure 2.3 *Tasks.xml file contains configuration scripts, which can be set to execute from topology definition files.*

Now let's look at a more complicated task – `SetNetworkLocationPrivate`. It is a little bit different from `EnableRemoteDesktop` that we already discussed, and has a set of additional XML elements – `<Copy/>`. Copy part of configuration task is different from execute, it copies the files that are required for script execution during VHD preparation stage. If your configuration task requires external tools or data files that are not on original sys-prepped VHD, you need to copy them to VHD before VM deployment, because there is no guarantee, that the deployed VM will have access to the file shares that contain those tools at the time of OS deployment, once the VM comes alive.

The most interesting aspect of configuration task infrastructure is the ability to survive system reboot, something that is not supported by unattend.xml originally. If the reboot occurs after the first logon, unattend.xml is no longer analyzed by OS, and all tasks that are defined in it after the task that initiated reboot will be simply ignored. To work around this we use another technique that makes the OS to execute tasks on logon, calling the task from the RunOnce registry key. Here is how it is done:
The task that can potentially require reboot should be decorated with Reboot attribute, which can take three values – `Never (Default)`, `OnRequest`, and `Always`, e.g.,

```
<Task Name="SetAutologonAndReboot" Reboot="Always">
```

Specifying Reboot="Always" in the Task element causes the generated script file to automatically append the lines below:

```
reg add HKLM\Software\Microsoft\Windows\CurrentVersion\RunOnce /f /v 10-
ShowFileExtensions /d "%SystemDrive%\Deployment\10-ShowFileExtensionsTask.cmd >
%SystemDrive%\Deployment\10-ShowFileExtensionsTask.log 2<&1"

echo --- Rebooting in 30 seconds.

shutdown /r /t 30
```

These lines put the call to execute the next configuration script into RunOnce, and call shutdown with the delay of 30 seconds. Although the configuration tasks are meant to be synchronous, they can still trigger some process within operating system that will take time to complete. The delay is specified so the OS gets adequate time to "digest" the current configuration task.

In case, the value of Reboot attribute is a conditional `OnRequest`, a slightly more complex code is added to the end of script, which analyzes the ErrorLevel after the task execution and makes a decision of whether to reboot system or to call the next script in the chain directly.

Note that topology definition file in its current form cannot be constrained by an XML schema, as the names of XML elements corresponding to tasks are not predefined and their set can be expanded by adding more task definitions to *tasks.xml*. This is not a conventional way of using XML. The reason we chose to put task references in this way is to reduce the amount of text in topology definition and improve readability. For example, the following line

```
<AddUserToLocalAdministrators UserName="mstest\msuser"/>
```

is easier to read than

```
<Task Name="AddUserToLocalAdministrators">
     <Parameter Name="UserName" Value="mstest\msuser"/>
</Task>
```

At the same time it makes XML impossible to conform to a schema and hard to validate with conventional XML tools. It also complicates discovery. So it is a trade-off and once the tool is made public we may reconsider this design decision based on the community feedback.

2.5 Defining configuration tasks

Once started the deployment will go through the following stages:

1. Factory configuration data is loaded from the corresponding xml files.

2. Topology definition is loaded from the specified xml file.

3. Task references in topology definition are parsed and separated into two phases – Copy and Execute. Copy phase is when external file(s) required to execute tasks are copied from specified location (usually a network share on a corporate network) to a specified path on a VHD. Execute tasks are formed into an execution chain, which starts with an unattend.xml. If any task in the middle requires reboot to continue, then the next task execution is placed into RunOnce registry key, to be executed after reboot. The RunOnce creation code is automatically injected into the task script, if the task declares that conditional or unconditional reboot is required.

4. Factory searches for OS setup media. The media can be in three formats: ISO, WIM or VHD. WIM media is part of the common distribution media package. If you have OS setup on a DVD, you can find the *install.wim* file under *sources* folder. The media is located in the media storage. Media storage is a set of shared folders containing released OS media and OS media currently in development. We use a combination of XML mapping file and Windows PowerShell script to locate and retrieve proper OS media based on specific description, e.g., Architecture=AMD64, Platform=Windows Server 2008 R2, SKU=ServerDataCenter, Locale=en-US.

5. The VHD media is optionally copied to a VHD cache folder to accelerate subsequent access, if VHD cache folder is defined during initial configuration of the factory.

6. Factory looks for a host machine. Physical hosts are selected only if topology defines machine as `<Physical/>` or if no available virtual hosts are found. Otherwise, the virtual host is selected. If there are several virtual hosts defined, the factory will pick up the first one that is available and has enough free memory and hard drive space. The minimum requirements for free memory and hard drive space are defined in factory.xml file. The memory should be enough to deploy the machine according to specs and still have amount of memory defined in ReservedHostMemoryMB attribute to avoid overcommitting memory on the host system.

7. A random name for machine is generated. Custom name generation schema can be defined. For example, default is `Schema= 'BNN-AADDD', Base='Server'`, where B stands for a base part, NN is an incremented counter for machines deployed in a particular topology, AA is a randomly generated literal of two Latin letters, DDD is a randomly generated literal of three decimal numbers. For a topology defined in Figure 2.2, the machine names will look something like *Server01-ZH567* and *Server02-ZH567*.

8. VHD is copied to selected host. It will serve as a base VHD for all machines that specify media matched by this VHD.

9. Differencing VHD is created for the base VHD.

10. Remote CredSSP session is established with selected host. In these session the following tasks are executed:
    a. Diff VHD is mounted.
    b. Unattend.xml, which was prepared in stage 3, is copied to VHD root.
    c. Files defined in Copy section of the referenced task definitions are copied to specified folders on the mounted VHD.
    d. Configuration scripts are copied to the mounted VHD.

11. Differencing VHD is dismounted.

12. For a virtual deployment, VM is created and started with the diff VHD attached to it.
    For a physical deployment, multi-boot entry is created, pointing to a differencing VHD, boot configuration data (BCD) is updated to automatically boot from this VHD.

13. For a virtual deployment, VM is started.
    For a physical deployment, physical host machine is restarted and set to automatically boot into VHD starting OS deployment after restart.

14. Text file containing the names of the deployed machines is generated to be potentially consumed by other automation infrastructure.

15. RDG file is generated to use with Remote Desktop Connection manager tool freely available from Microsoft downloads site, (Microsoft 2010).

Simplified view of this workflow for the case of VM deployment is given on Figure 2.4

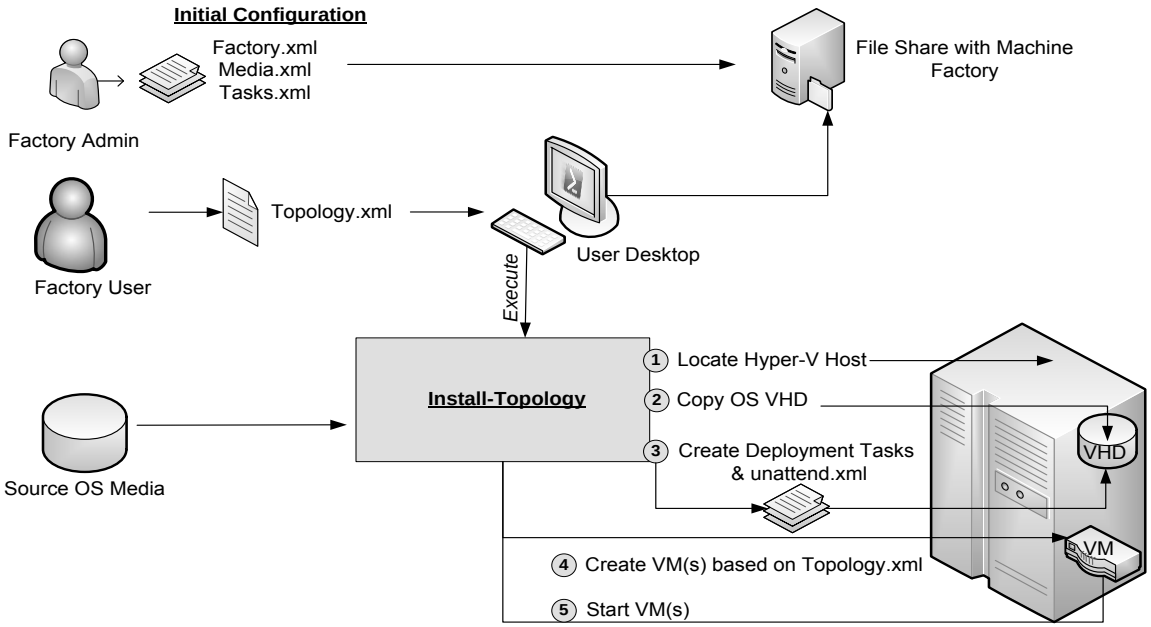


Figure 2.4 *VM deployment process.*

After VM or physical machine starts, the deployment is controlled by *unattend.xml* and, in case one or more reboots are requested by configuration tasks, by RunOnce registry setting.

2.6 A more complex deployment scenario

As you can see, the machine factory can be easily configured to solve virtually any automated deployment scenarios. Once you have figured out what kind of scripts or command-line tools you need to run to configure your system, you can put them to Tasks.xml and reference them in your topology definition file.

The automation does not necessarily have to be limited only to deployment and configuration of the test topologies but can incorporate test execution and reporting as well. See Figure 2.5 for an example.

```
<?xml version="1.0" encoding="utf-8"?>
<Topology Name="SMD">
    <Description>A set of server SKUs for SMD testing.</Description>

    <DefaultBranch>branch_main</DefaultBranch>

    <SetGlobalVariables>
        $global:BinPath = Get-LatestAvailablePath -Branch $defaultBranch -Path
'amd64fre\bin'
        $global:TestPath = Get-LatestAvailablePath -Branch $defaultBranch -Path
'amd64fretest\bin\TEST\ADMINTEST'
    </SetGlobalVariables>

    <RunOnFirstLogon>
        <SetNetworkLocationPrivate/>
        <EnablePing/>
        <EnableRemoteDesktop/>
        <EnableDebugging/>
        <JoinDomainWithRetry DomainName="mstest.microsoft.com"
UserName="mstest\msuser" Password="passw0rd"/>
        <AddUserToLocalAdministrators UserName="mstest\msuser"/>
        <SetAutologonAndReboot DomainName="mstest" UserName="msuser"
Password="passw0rd"/>
        <SetConsoleProperties/>
        <DeployServerManagerSymbols OSBinRoot="$BinPath"/>
        <DeployServerManagerTests BinPath="$BinPath" TestPath="$TestPath"/>
    </RunOnFirstLogon>

    <Group Name="MSTest" UserName="mstest\msuser" Password="passw0rd"
StoreAsClearText="True"/>

    <Machine Name="Server1" Count="1"/>
        <MemorySizeMB>1024</MemorySizeMB>
        <DiskSizeGB>60</DiskSizeGB>
        <OSArchitecture>amd64</OSArchitecture>
        <Platform>Windows Server 8</Platform>
        <SKU>ServerDataCenter</SKU>
        <OSBaseLanguage>ps-ps</OSBaseLanguage>
        <RunOnFirstLogon>
            <RunSMDTestsFTP Arguments="/ui"/>
        </RunOnFirstLogon>
    </Machine>

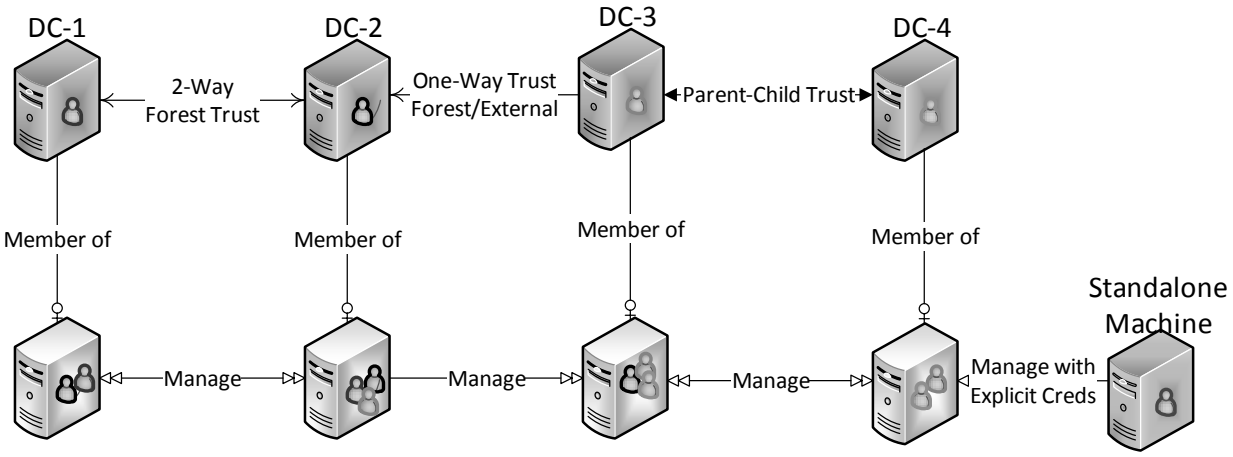
    <Machine Name="Server2" Count="1">
        <MemorySizeMB>1024</MemorySizeMB>
        <DiskSizeGB>60</DiskSizeGB>
        <OSArchitecture>amd64</OSArchitecture>
        <Platform>Windows Server 8</Platform>
        <SKU>ServerDataCenter</SKU>
        <OSBaseLanguage>ps-ps</OSBaseLanguage>
    </Machine>

</Topology>
```

Figure 2.5 Topology that automatically executes full test pass for UI components of Server Manager

The complexity of topology can also be quite high and can involve deploying Domain Controllers, setting forests and domain trust relationship. On Figure 2.6, you can see a fairly complex topology that we have used for cross-domain server management testing.

MULTI-DOMAIN TRUST TOPOLOGY DEPLOYMENT USING MACHINE FACTORY



- Multi-Domain trust topology defined using VM factory XML constructs
- Automated setup for domain controllers and domain members.
- Hybrid Forest/External trusts defined and setup between domains
- Domain Admin, Domain Users and Local admin/user accounts created unattended
- Automated setup for Two-Way, One-way, Parent-Child, Transitive and Non-Transitive trusts.

Legend		
Topology Elements		
Symbol	Count	Description
	11	Domain Controllers, Domain Members and Standalone machines
	18	Admin Users on Machines

Figure 2.6 *Multi-domain hybrid-trust topology*

2.7 Troubleshooting and error reporting

First step to identify potential issues is to run `Install-Topology <topology-definition>.xml with -WhatIf` switch as described in 2.3, which will detect problems with xml data files and also host connectivity problems.

After this you can run the same command without –WhatIf switch and monitor the console output. The critical errors in the script execution will cause script termination as shown on Figure 2.7. Non-critical errors are written to console output to be analyzed later.

If you see a non-critical error during script execution and feel that this is something that is going to affect topology deployment in a bad way, you can hit Ctrl-C to terminate execution at any point.

Or if you identified the problem only after script execution finishes, you can use the console output to analyze at which point the problem happened, then set breakpoint in a proper place and debug the issue in Windows PowerShell ISE (a PowerShell host application that has interactive debugging capabilities).

The problems that occur during unattended configuration phase, when VM or physical machine come alive, can be identified by looking the logs produced by configuration scripts, which are placed in *Deployment* folder of the system drive of the deployed machine.

```
PS D:\MachineFactory> Install-Topology .\Topologies\TopologyClient2.xml -verbose
Deployment started - Saturday, June 23, 2012 6:19:55 PM
Loading factory configuration.
Loading topology data.
Loading tasks.
Loading media.
Generating 1 machines for 'Client'.
Looking for VHD for 'Client' machine definition.

< … part of the output skipped … >

Create base differencing VHD if necessary (one per topology/machine definition):
D:\VHD\RSAT-Client-dbdde144-67ce-4cc9-80c3-2d6f024ab4df.diff.vhd
Generating base differencing VHD D:\VHD\RSAT-Client-dbdde144-67ce-4cc9-80c3-
2d6f024ab4df.diff.vhd for parent VHD D:\VHD\client_enterprise_en-us.vhd on server
MAX-WS-01.

Test-WmiResult : Failed to create the virtual hard disk.
The system failed to create 'D:\VHD\RSAT-Client-dbdde144-67ce-4cc9-80c3-
2d6f024ab4df.diff.vhd': The process
cannot access the file because it is being used by another process. (0x80070020).
At D:\MachineFactory\Modules\HyperV\Disk.ps1:920 char:22
+                 if ((Test-WmiResult -result $result -wait:$wait -JobWaitText ($l
...
+ ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
    + CategoryInfo          : NotSpecified: (:) [Write-Error], WriteErrorException
    + FullyQualifiedErrorId :
Microsoft.PowerShell.Commands.WriteErrorException,Test-WMIResult
```

Figure 2.7 *Sample script execution output, which contains a critical error.*

3 Plug-ability and Automation Streamlining

Machine factory provides a standalone solution for provisioning highly-configured machine topologies. This loosely-coupled approach also makes it a very plug-able component in automating the test execution/reporting ecosystem. The flexible ends of XML-based input and output make it very consumable for any application programming interface (API).

3.1 Machine Factory Interfacing

Whether providing a web-based interface or hooking-up a UI application front-end, the provisioning system fits conveniently in the back-end with its flexible XML-based interfacing. In fact, it can very easily be tweaked into a type-safe software module if adapted to the .NET object-model. With Windows PowerShell at its core, this capability is readily available and simply begs adoption.

The rich interface of HTML5 with the power of JavaScript, applied as the front-end to machine factory, can provide a complete web-based solution sufficing the needs of any test environment. This is not implemented yet, but we plan it for future development later this year.

3.2 Plugging in execution engine

The XML topology definition coming out of machine factory can easily incorporate any execution framework. In fact, we have applied solutions which complete the automation spectrum starting from topology definition, provisioning, invoking a test execution engine (Markin 2010) and doing automated reporting. To make our automation complete, we set Windows Task Scheduler to invoke test execution cycle on a daily basis. In that way machines are deployed, test pass executed and reports are sent to subscribers every day with no human interaction.

3.3 Automated Reporting

The final piece of the automated test-execution environment is reporting. The request for test execution, which started from a machine topology definition, was provisioned by the machine factory and the output was consumed by an execution framework (incorporating a variety of test harnesses). The resultant output of this process is a test-result log collection (in varying formats) along with the original XML topology definition.

The last piece of automated reporting culminates the process by implementing two components. The first component is a parser having the capability to map test-logs to registered log formats. This component parses the logs, can optionally convert the results into predefined strongly-typed result objects (which can optionally be passed on to a trend-analyzer), and saves them in a database. The second component is a monitoring service which detects new results in database, processes them into tabular reports and publishes them on a website or through email.

```
Logs: \\ReportsDB\Triage\SFD\FAILED_2012-03-14-062824
Build: Win8RTM.amd64fre.111118-2330

Pass Percentage:    86%
Tests Executed:     126
Tests Passed: 109
Tests Failed: 17
```

Feature	Pass	Fail	Error	Skip	Total	Pass Rate	Time Taken (Minutes)	Test Owner
BVT	7	0	0	0	7	100%	8	mikeS
Content	1	0	0	0	1	100%	1	bilalK
Services	44	12	0	0	56	79%	37	rogerp
PerfCounters	41	0	0	0	41	100%	19	markA
PluginManager	13	2	1	0	16	81%	11	krisd
WMIProvider	3	2	0	0	5	60%	1	ginaM

Figure 3.1 *Sample Test-Pass Report*

A logical extension to this process would be an analyzer module to detect failure-trends and pattern-match existing issues/bugs. The newly detected issues could be logged in a bug database and already existing issues could have their tally incremented.

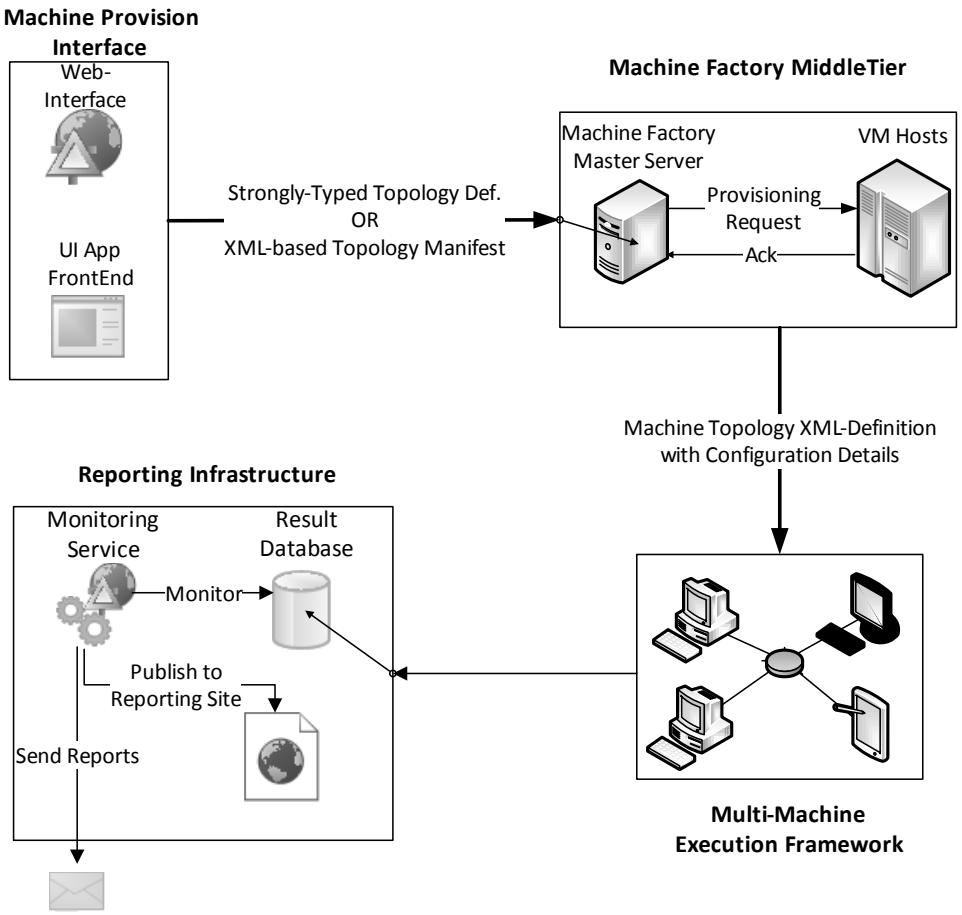


Figure 3.2 *End-to-End Provisioning, Execution and Reporting Process*

4 Why you might consider using Machine Factory

There are quite a few solutions that exist in the market today ranging from shrink-wrapped software products such as VMWare vCenter Lab Manager (VMWare 2012) or Convirt Solutions: QA & Test Labs (Convirture 2011) to IT consultants that build customized QA automation systems. However, most of these solutions end up being expensive not just in terms of procurement and training but also maintenance and adaptability to complex test topologies.

The Cost factor: Machine Factory is a flexible yet completely free tool that does not depend on any technology other than the components that come with a standard Windows OS and it addresses the same need for provisioning of complex multi-machine test environments. Machine Factory uses prevalent technologies that are native to Windows OS and is built on top of Windows PowerShell which has a strong community and following making it easy to customize and maintain. On the other hand, you could end up spending upwards of $15K for software solutions such as the VMWare vCenter Lab Manager (VMWare 2012) in addition to ongoing maintenance and training expenses.

Increased productivity: As a case in point, the use of Machine factory enabled our team to deploy a 100 machine test environment with complex configuration in a matter of 30 minutes reliably on a consistent basis as opposed to 5 days of manual effort every time getting physical hosts configured with Hyper-V, customizing OS images and performing post-deployment configuration. The complete system including execution and reporting enabled the team to execute integration tests on complex cross-domain

topologies on a weekly basis as opposed to once in a project milestone resulting in faster identification and resolution of software quality issues.

5 What you can learn from it

Machine factory is a powerful tool by itself, but what adds to its value is that it can be considered the showcase of different management technologies, and a set of real-world usage examples to automate your quality assurance process and make the life of a test engineer easier. Whether you adopt it as yet another test tool in your arsenal, or cannibalize it for parts in your new, super awesome test infrastructure, you will learn how to use those technologies and become more efficient in what you do.

As the tool is 100% PowerShell script, it does not require any reverse engineering to understand how it works, which makes it easy to maintain and adapt.

Here is an extensive and yet incomplete list of technologies that we have used in Machine Factory to automate multi-machine topology deployment.

Windows PowerShell	Windows PowerShell orchestrates media retrieval and preparation, and machine provisioning. On this example, you can learn how to create fairly complex PowerShell modules and create automation tools for your test infrastructure. (Microsoft, Windows Powershell, 2012)
Windows Remote Management	Windows PowerShell Remoting on top of Windows Remote Management protocol is used to communicate with remote hosts.
COM	COM APIs are invoked in some configuration tasks, e.g., setting network location to Private.
.NET	.NET APIs are invoked all across the PowerShell cmdlets in this module. Examples are XML read using XML DOM, RDG file generation using XMLWriter, Secure String encoding, etc.
BITS	Background Intelligent Transfer System (BITS) and RoboCopy tool is used for fail-safe copying of media and other files.
Hyper-V	Hyper-V is used to provision virtual machines. Management of remote Hyper-V hosts is done using WMI.
WMI	Used to check available memory and hard drive space on remote hosts, communicate with Hyper-V and BCD (Microsoft, Windows Management Instrumentation, 2012)
VHD Boot	Used to provision physical machines.
Boot Configuration Data (BCD)	Used to set machine to boot from VHD.
Windows PowerShell Management Library for Hyper-V	A Windows PowerShell module containing cmdlets to execute Hyper-V operations on a local or remote machine via WMI interfaces. (CodePlex 2009)
WIM to VHD Converter	A tool used to convert ISO and WIM media to VHD. It is freely available for download from Microsoft code gallery site: WIM to VHD converter for Windows 8 (Microsoft, Script Convert-WindowsImage.ps1 - WIM2VHD for Windows 8!, 2012) or WIM to VHD converter for Vista and Windows 7 (Microsoft, WIM to

	VHD converter for Vista and Windows 7, 2012)
Unattend.xml	Used to automate deployment and configuration during OS setup and first logon phase.
RunOnce	RunOnce registry key which is used to automate machine configuration across multiple reboots
CMD scripts	CMD scripts are used for execution of configuration tasks. CMD scripts can optionally call PowerShell or VBS scripts for more complex configuration tasks.
Wmic.exe	Command-line tool to invoke WMI methods in case Windows PowerShell is not installed on the target machine.
Autologon	Used to give the deployed machine the ability to log on automatically if configuration requires one or several reboots.
CredSSP	CredSSP authentication mechanism allows access to remote share from within remote session.
VHD Differencing	Technology that allows creation of several VHDs based on one VHD to conserve disk space and minimize copy time. You can deploy multiple machines from the same base sys-prepped VHD.

6 Frequently asked questions

How long does it take to get the different VMs fully configured?

It takes several minutes for simple deployment on a mainstream datacenter machine, but can extend up to 1 hour or even more, depending on whether the base VHD file is already copied to the host, the amount of configuration required, the host's 'horse power' (RAM, HDD speed, CPU), network bandwidth (can affect VHD copy time), etc. One example is a *100 machine* test environment that we used for scalability and performance studies. It takes about 30 minutes to deploy 100 machines with the same configuration on 5 HyperV hosts. Script execution takes even less than that, since script execution is finished once all VMs are started.

Are you taking shortcuts, using the Virtual Disk technology? Does EVERY VM have to be a fully configured OS from initial install, or are there shortcuts to take?

We use several techniques to optimize performance. For example, we use so called *sysprepped* VHDs – OS already installed and just a few extra steps are required until the first logon boot – setting drivers, defining computer name and built-in administrator. We also use *VHD Diff* – the VHD, which contains the OS. is copied to the host machine only once, and the actual deployment uses differencing VHD, which uses base VHD file, but all changes are written to a diff VHD file.

What are the pieces that take the most amount of time? How long are they?

Most of the time is spent copying operating system VHD to the VM host over the network, approximately 5 minutes. In case the VHD is already cached on VM host, this time is not spent. Another significant portion of time is taken by copying files to the diff VHD that will be used for the actual deployment. In some of our use cases, we need to copy around 10GB of data onto the VHD to be used for our testing. If the amount of data that you need on the deployed machine is small, then this time will be around several seconds. Note that if you have several machines with the same configuration and data, e.g.,

```
<Machine Name="Client" Count="10">
       <MemorySizeMB>1024</MemorySizeMB>
       <OSArchitecture>amd64</OSArchitecture>
       <Platform>Windows 8</Platform>
       <SKU>Enterprise</SKU>
       <OSBaseLanguage>en-us</OSBaseLanguage>
</Machine>
```

Then, the actual copying will be done only once and several diff VHDs will be created from the VHD that has the copied data - a process that takes only as much as several seconds.

Once the VM starts to boot, the script execution is finished. We do not control or get the notifications of the deployment progress. At the same time we understand that tracking deployment progress is a nice-to-have feature and we are currently investigate two technologies that can be used for this – one can only be used for VM deployment progress, but does not require the VM to be on the same physical network, other can be used for both VM and physical machines but requires machines to have access to the same physical network.

How long is each VM active in its configuration, before it is destroyed or rebuilt?

The lifetime of VM can be anything from several hours to weeks or months. For example, to compare performance of the certain features of the new Windows Server OS with the old Windows Server, we are deploying a set of machines with the new Windows Server installed every week, and our organization uses it for scenario verifications. There is also a set of machines, which have old Windows Server installed, deployed more than one year ago, and it is used for comparison and to test possible performance degradations.

7 Future work

The goals behind the development of machine provisioning tool were always two-prong:

- Easy solution to deploy small pre-configured set of VMs for a single consultant/developer/tester using his notebook or desktop
- Enterprise-level automation to deploy large quantities of machines for production QA.

Goal 1 is achieved but requires further improvement to simplify user experience and reduce learning curve.

Goal 2 is achieved for some applications like performance, stress and scalability studies. It still requires some web-based frontend UI and web-server backend to be added to the current infrastructure for this to be usable in enterprise environment, where users may not have PowerShell skills required to use the scripts directly.

As the internal adoption of the tool continues, we are planning to make some modifications to the tool to make it available to general public. Right now, it is largely oriented on Microsoft Windows Server team and some parts of it are dependent on Microsoft internal infrastructure. We need to refactor those internal dependencies to become configurations and extensions, rather than embedded in the scripts.

Another work that is ongoing is to improve the logging and debugging – mostly around *WhatIf*, *Verbose*, *Debug* support (those are proprietary Windows PowerShell technologies that are better than traditional console output used in regular command-line tools). We plan to finish this work in September, which will allow us to publish the tool on the CodePlex site.

8 Conclusion

It is important to understand that the main value proposition of the presented tool is not to provide an automated way to deploy virtual machines. This problem has been successfully addressed by a number of products and tools that are currently in existence. The main challenge is usually not to deploy machines, but to configure them afterwards to provide a staged multi-machine environment, ready for manual scenario validation or automated test execution. There are quite a few existing tools in the market that solve the problem but they are quite complicated and require complex deployment supporting infrastructure and use of proprietary execution agents. In addition to the time and expertise required to operationalize, costs for these tools can range anywhere upwards of $1000.

When we created our tool, we intentionally designed it to be as simple and transparent as possible, basically just a thin script-based overlay over existing well-known technologies, such as *Unattend.xml* and *RunOnce*. That helped the initial adoption of the tool and contributed to its success.

For main deployments, we are currently using 15-20 HyperV servers in our data center, each capable of hosting 20 virtual machines with 2GB RAM allocated to each guest. The deployment of 100 machines on 5 HyperV servers takes less than 30 minutes with no human interaction whatsoever, except initial launch of *Install-Topology* command. Previously, the same operation took hours or even days to fully deploy and configure those machines and required substantial human resources.

The development of the system took one man-month from idea to a working prototype and some part-time effort to expand and improve the tool during initial adoption. The tool does not require any supporting infrastructure so there are no dedicated resources to maintain it. A single machine running *Windows Server 2008 R2* is an entry-level hardware that you need, and initial configuration requires reading one page of *Get Started* document and 5 minutes spent to set initial configuration.

Windows, along with its built-in technologies like Windows PowerShell, WMI, BITS and Hyper-V, are the building blocks we used to accomplish this goal. Our tool is an example of how these technologies can be used to address a wide variety of automation problems, such as machine provisioning, configuration, test execution and reporting.

There are several ways that the tool can be used as can be seen on the example of our organization:

1. Back up machine provisioning tool in case of the main provisioning system downtime or maintenance.
2. Scalability, stress and performance studies, which require rapid deployment of hundreds of pre-configured machines with minimum manual effort.
3. Repeating deployments of complex topologies, e.g. multi-domain hybrid trust topology shown in Figure 2.6.
4. Fully automated execution of test passes – currently also a ‘backup’ system for the main test execution lab.
5. Deploying machines with different SKUs, locales, architecture, configuration for bug bashes.
6. In-office virtual and physical machine deployment for private test runs, scenario validation, etc.

As can be seen from this list, on many occasions we use this system as a ‘backup’ system when our primary, enterprise-wide system (which provides additional capabilities, such as self-service VM checkout) is down for any reason. Due to its simplicity and flexibility, it can be used even in the unlikely case of datacenter’s complete unavailability, since it can still employ office machines or personal notebooks for small scale deployments. Our system also provides a “stop-gap” solution on many occasions, where the main system is not flexible or capable enough to be used.

9 References

MICROSOFT CORPORATION. 2012, Windows PowerShell
http://technet.microsoft.com/library/bb978526.aspx (Updated: February 15, 2012)

MICROSOFT CORPORATION. 2012, Windows Management Instrumentation
http://msdn.microsoft.com/en-us/library/windows/desktop/aa394582(v=vs.85).aspx(accessed August 10, 2012).

CodePlex 2009, Windows PowerShell Management Library for Hyper-V
http://pshyperv.codeplex.com/ (Last edited Oct 31 2009 at 5:13 PM by jamesone, version 11)

MICROSOFT CORPORATION. 2010, Remote Desktop Connection Manager – RDCMan 2.2
http://www.microsoft.com/en-us/download/details.aspx?id=21101 (accessed August 10, 2012).

MSDN 2012, WIM to VHD converter for Vista and Windows 7
http://archive.msdn.microsoft.com/wim2vhd (Last edited May 31 at 3:49 PM by mikekol, version 21)

MICROSOFT CORPORATION. 2012, Script Convert-WindowsImage.ps1 - WIM2VHD for Windows 8!
http://gallery.technet.microsoft.com/scriptcenter/Convert-WindowsImageps1-0fe23a8f (accessed August 10, 2012).

VMWARE INC. 2012, VMWare vCenter Lab Manager, Software Testing Lifecycle Automation for Developers and QA
http://www.vmware.com/products/datacenter-virtualization/labmanager/overview.html (accessed August 10, 2012).

CONVIRTURE 2011, Convirt Solutions: QA & Test Labs
http://www.convirture.com/solutions_qa.php (accessed August 10, 2012).

Markin et al. 2010, Test Environment Configuration in a Complex World
http://www.uploads.pnsqc.org/2010/papers/Markin_French_Hong_Test_Environment_Configuration_in_a_Complex_World.pdf

Automating Build Installs Using AutoIt

Brian Yoss
Brian_Yoss@McAfee.com

Abstract

How do you know which build to use for testing? Where do you look on the network once you know which build to use? Do you use the debug or release version? What install options do you use? How long is this going to take you to get your testing environment up and running? These are all questions that can easily be answered with the help of free scripting tool called AutoIt. This paper will provide you with an understanding of how AutoIt can be used to build an automated GUI utility to help QA acquire the correct build for testing and to get it installed in their environment with a few clicks of the mouse.

On my team at McAfee, it has been a challenge for QA to stay on top of builds delivered from our configuration management system. Depending on the environment, builds could be delivered hourly, daily, weekly, etc. Once QA determines what build they need to use, they need to find it, copy it to their local testing environment, unzip it, and then install it. The actual install process can be daunting and require multiple steps. This can add up to time lost actually testing the product due to the entire build and installation management overhead.

This paper describes possible ideas and processes to be considered when developing an automated utility to produce a streamlined QA testing environment. Further, this type of automation can create testing efficiency for a routine but necessary step, thereby reducing the overall cost to your organization.

Biography

Brian Yoss is a Senior Software QA Engineer at McAfee. He has over 10 years of QA experience and has been at McAfee for 7 years and the QA Lead on McAfee's flagship enterprise security management product for the past 5 years. Brian graduated from Oregon State University with a BS in Business, with an emphasis in Management Information Systems and a minor in Communications.

1 Introduction

I have always had an interest in programming and automation but wasn't quite sure where to get started. I don't have a Computer Science degree, but I know there are tools out there that are easy enough to use without a specific degree that teaches you how to be a programmer. During early 2007, I expressed my interest in automation and was given the opportunity to shadow our automation engineer to see how he did things. While I was learning about some of the processes, concepts, and ideas regarding automation, I was shown a very nice free scripting language called AutoIt. At that time I thought it could be very useful in some of the things we were trying to accomplish, but I didn't have any specific ideas on how I could use it. Fast forward a few years and I finally had some ideas on how I could help improve efficiency and decrease overhead and it involved using AutoIt.

This paper describes my story of looking for QA inefficiencies and building a tool that would be beneficial for not only QA, but other departments within the organization.

Date	Milestone
December 2007	AutoIt automated installer script delivered for Build Verification Test
December 2009	Cluster batch file completed for environment setup
February 2012	ePolicy Orchestrator Install Tool project start
March 2nd, 2012	ePolicy Orchestrator Install Tool Proof of Concept delivered
March 13th, 2012	ePolicy Orchestrator Install Tool Version 0.5 delivered
March 28th, 2012	ePolicy Orchestrator Install Tool Version 1.0 delivered
April 12th, 2012	ePolicy Orchestrator Install Tool Version 1.1 delivered
May 21st, 2012	ePolicy Orchestrator Install Tool Version 1.3 delivered

Table 1: Project Milestone Timelines

2 Build Verification Tests

2.1.1 Problem

Over the past 7 years, I have been involved with testing our enterprise management product called ePolicy Orchestrator (ePO). During December 2007, we were developing a set of automation tests called the Build Verification Tests (BVT) and there was a need to have our product installation automated. The automation tool that we were using at the time didn't have a nice way to automate our product installs and I thought this could be a good opportunity to give AutoIt a chance.

The main pieces that I had to incorporate into the script were:

1) Have the script automatically walk through the installer from start to finish
2) Have the script configurable to allow different settings within the installer

2.1.2 Solution

At a very basic level, AutoIt has a record and playback feature which allowed me to record the exact steps through the install wizard that I needed to use. Recording the steps that I performed allowed me to see how AutoIt inserted the code into the script it created. Once I could see the code, I was able to manipulate it to do exactly what I wanted. After my script was complete, I converted it to an executable file and then we were able to call that file to install our product before our automated BVT was run.

Because of the different environment setups we needed to test against, I needed to make my script configurable to allow different settings during the install process. AutoIt allows configuration files to be

used in conjunction with the script. These files contain different settings that the script reads on execution to allow the script to change what settings are used in the installer.

2.1.3 Drawback

The drawback to this approach was that there were timing issues when clicking through the installer interface. There were a lot of pauses that needed to be manually added to the script in order to address any inconsistencies with clicking through the install wizard. There was also the problem with specific object ID's changing in the installer while we were developing it. If I was looking for a specific object ID to click on and that ID had changed, the installer would hang and would fail. This tool worked for what we needed, but it still required a good amount of overhead and there had to be a better way.

3 Cluster Testing

3.1.1 Problem

ePO is a product that is cluster aware and we support failover capabilities. I was given the task to ensure our product functioned correctly in a clustered environment. Building a clustered environment for testing is a very daunting task. Assuming I already have the virtual machines available for use, some of the tasks involved are:

1) Setting up at least 2 systems to act as active and passive nodes. This typically takes about 30-45 minutes for each node.
2) Setting up a storage server for the application and cluster data which takes around an hour.
3) Setting up a remote SQL server which can also take about an hour.
4) Installing the Cluster Service in the environment which takes about 30 minutes.
5) Installing ePO which takes about 30 minutes.

Because of the amount of setup that is involved when setting up a clustered environment, I thought of ways that I could automate the process in order to lower the amount of overhead involved. The initial time for setup would be the highest cost, but once it is all configured properly, the future overhead is very low.

During December of 2009, I sat down and determined that the main pieces that I had to incorporate into the script were:

1) Allow the user to specify a build, package, and build type to install
2) Have the script revert VM snapshots and power them on
3) Have the script configure the cluster settings
4) Have the script find the build specified and copy it to my test system
5) Have the script unzip the build
6) Have the script install the build

3.1.2 Solution

After some initial research, I discovered that the majority of the tasks I was trying to accomplish could be completed via a command line. I was using virtual machines that I could call specific snapshots, power on, and power off from the command line. Our product could be installed silently via a command line. I could even configure my cluster environment using the command line. I had some experience in the past with building batch files, and I thought this would be something I could start with and build upon.

I was able to successfully use my batch file to build my cluster environment whenever I needed it. I cut down my environment setup time from about 4 hours to 45 minutes. There was a lot of interest in how I

created my batch file as soon as people heard about the setup efficiency because it could be used for any environment and not only for a cluster environment.

3.1.3 Drawback

I created documentation for others on the team to be able to create their own batch file. However, there was a lot of initial setup involved to get the batch file to run smoothly and it wasn't exactly user friendly. People wanted something easier.

4 ePO Install Tool (Proof Of Concept)

4.1.1 Problem

During Q1 of 2012, McAfee started tasking employees with coming up with "innovation" ideas within the organization. I thought this was a perfect time to take what I had done in my batch file and use AutoIt to build a nice Graphical User Interface (GUI) that was very user friendly so anyone could use it. I knew I had to start with a proof of concept (POC) and build it out from there to see what I could do. I wasn't sure what the end result would be, but I was up for the challenge.

The main pieces that I needed to incorporate into the tool were:

1) Allow the user to specify a build to install
2) Allow the user to specify any network credentials that are needed for the install
3) Allow the user to specify if they wanted to install the product using our packaged SQL Express or point to an existing SQL Server
4) Allow the user to specify the admin password for the product being installed
5) Mask any password information that the user enters
6) Only allow the install to start after all required fields are filled out
7) Have the tool automate the process of copying the build locally
8) Have the tool automate the process of unzipping the build
9) Have the tool automate the process of installing any pre-requisites
10) Have the tool automate the process of installing the build

4.1.2 Solution

Since I hadn't used the GUI piece of AutoIt before, I spent some initial time researching the API to see exactly how to create the GUI. AutoIt has a wonderful set of web pages that have a ton of information and examples of how to create some amazing tools.

Once I figured out how to create a baseline GUI, I then had to determine what functions I needed to use to add the pieces of functionality that I determined were needed. I knew there were a lot of different configuration options that I could include, but I wanted to keep things simple at first to see if this was a feasible project.

Because there are different versions of the product that the tool needed to accommodate, I was able to keep the version specific information on their own tabs. The information that was generic across version was kept consistent for each version that was selected.

When the tool launches, it reads the current environment such as domain, user, and machine name and adds this information as default values to help the user. Each one of the pre-populated values can be changed based on the user's needs. Below is a screenshot of my POC and I sent this out to a select few within QA to get feedback and to address any issues that they found.

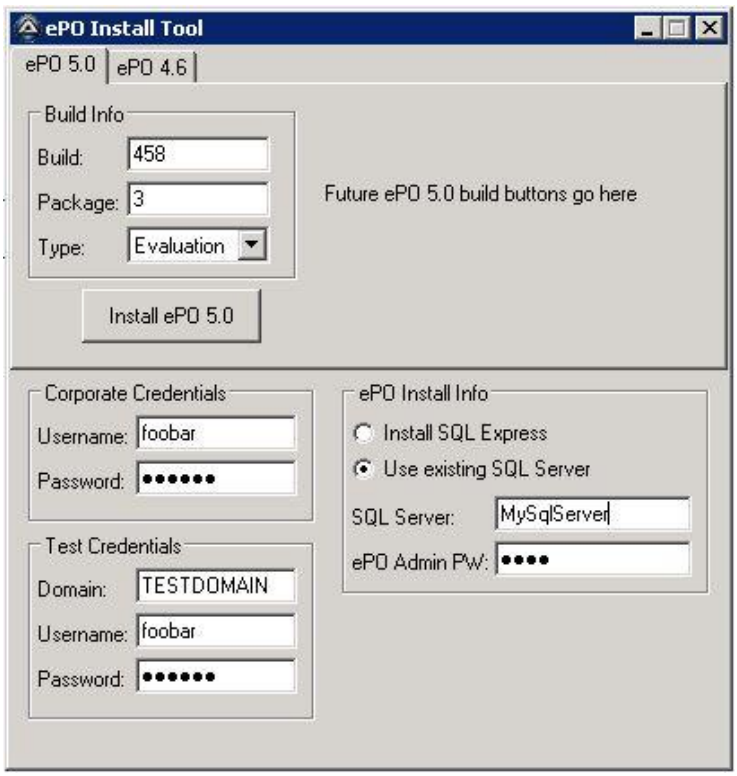


Figure 1: ePO Install Tool POC – 5.0 Tab

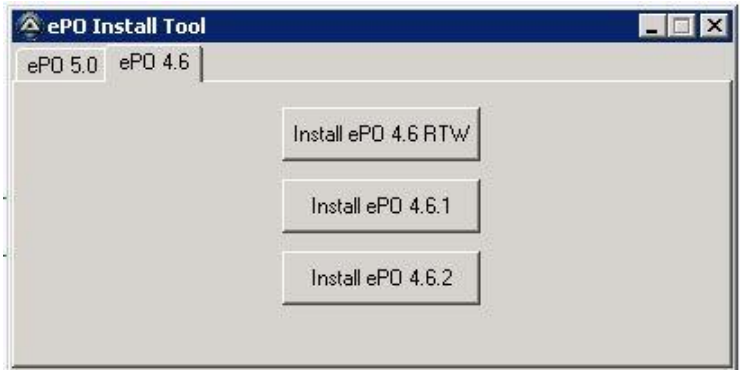


Figure 2: ePO Install Tool POC – 4.6 Tab

4.1.3 Drawback

This was a good start, but there were still a lot of features that could be added or improved upon. At this point I knew that my idea was possible, but I needed to add more pieces to really make this useful for others.

One of the main pieces of feedback that I received was to add a list of possible builds to install instead of needing to know the exact build, package, and type.

5 ePO Install Tool (Version 0.5)

5.1.1 Updates

Based on the feedback that I received from my POC, I investigated a way to provide a list of possible builds to install. McAfee uses a Configuration Management DB (eCM) that stores all of the information regarding our builds for every product. After some research, I discovered that AutoIt has built in functions to connect to SQL databases. I just needed to get the credentials to the SQL DB and then I could run a query against our ePO DB to return the results of our builds. After I had the results of our builds, I could pass that info into an array and populate a list box in the GUI to display the last 5 builds available in eCM. With the addition of a list box of possible builds to use, users could select a build to install or could still manually enter the build, package, and type if they chose to do so.

I also added a File and Help menu system that allowed users to exit the tool and to use help to navigate to an internal webpage that included information about all of the different pieces of functionality. After these updates were made to the tool, it was sent out to a wider audience that included QA, Development, and Technical Publications.

Below is a screenshot that displays the menu system and what the list box looks like so that users can see the available builds and their status. Selecting a build automatically populates the build, package, and type fields in order to install the build.

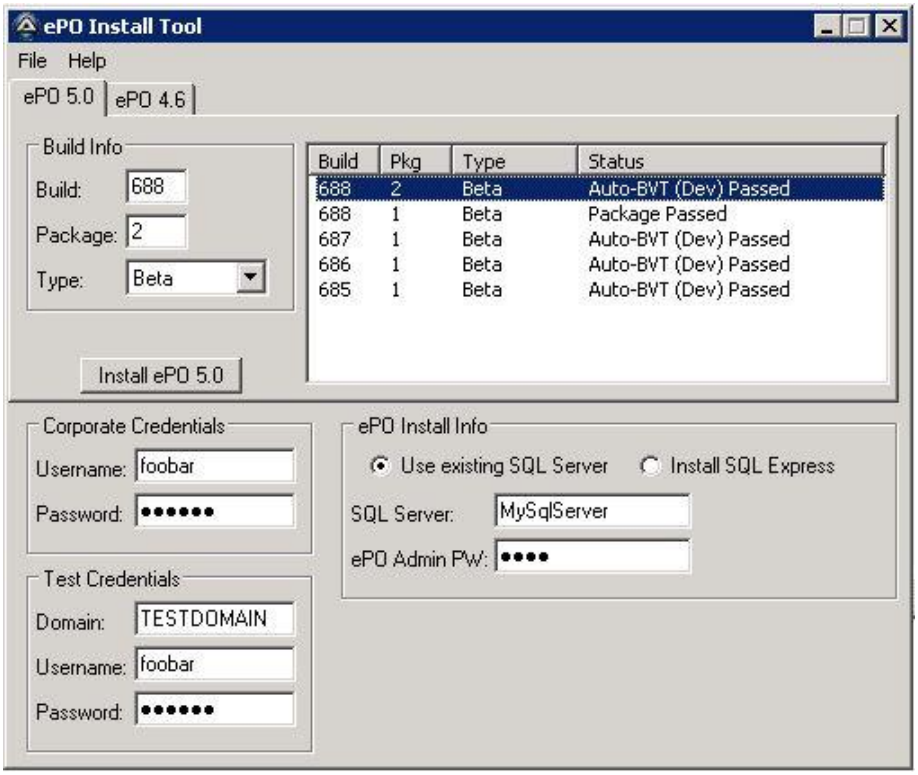


Figure 3: ePO Install Tool – Version 0.5

6 ePO Install Tool (Version 1.0)

6.1.1 Updates

During the release of Version 0.5, I started to receive a lot of feedback from users. QA, Development, Technical Publications, among others were all using the tool and they wanted to see more functionality. I even had people in the organization offer to help with updates since this wasn't my primary job and it was becoming something that required a lot of time to incorporate all of the updates that people wanted to see. I knew that there were still a lot of things that could be added to the tool, it was just a matter of finding the time to get it done. Some of the feature requests were:

1) I want to see more than the last 5 builds.
2) I want to install an Agent Handler too. Agent Handlers are a separate install that tie directly with ePO.
3) I want to be able to install builds with more versions of ePO.
4) I want to be able to provide the port that my SQL Server is using.
5) I want to be able to provide the admin name for my ePO install.
6) I want to be able to enter a license key.
7) I want to be able to provide the install path.

Based on this feedback, I started to update the tool. The first request was really easy because I just needed to change the SQL query from returning 5 to a higher number. I changed this to the last 20 builds, which I felt would suffice.

ePO has a piece of functionality called an Agent Handler, which is a separate installer. Users wanted to be able to use this tool to install an Agent Handler too. In order to install an Agent Handler, the user needs to provide the name of the ePO server that it will connect to. It also needs to know the admin name and password that ePO is using. These fields were added to the bottom of the tool in its own section labeled "AH Install Info."

During ePO development, there are a lot of different versions being developed and supported. Users wanted to be able to install all different versions of ePO. Tabs worked really well in this situation, so I added a new tab for each version that was being supported. Since ePO 4.6.3 was actively under development, I was able to use the SQL query to the eCM DB to pull the latest builds for that version as well.

Users also wanted to be able to specify the port that their SQL Servers were using. When the tool is launched, it defaults to the standard SQL port that we use. However, this port could be different based on their environment and users can change it for their specific needs.

In the previous versions of the tool, it only allowed the user to provide an admin password for the ePO install because that is the minimum info that is required for the admin. When the tool is launched, it defaults to the standard admin username that we use and users can now change that to what they need.

After an ePO install, users can enter a license key in ePO to provide access to different pieces of functionality. Instead of making users enter the license key manually after an install, they wanted to provide this info in the tool so it was done automatically. By default, the tool enters one of our standard testing license keys, but can also be changed to the users liking.

Because of all the different environment configurations that we need to test, users want to be able to install ePO to a specific path that they entered instead of using the default path. When the tool is launched, the default install path is provided and users can change it to their needs.

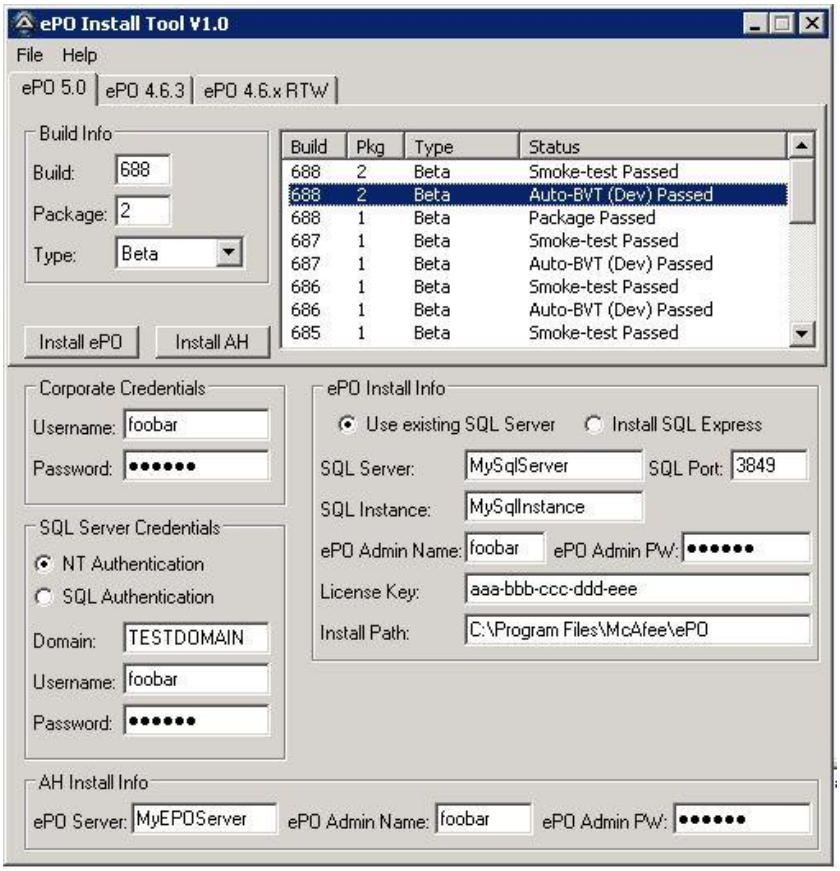


Figure 4: ePO Install Tool 5.0 Tab – Version 1.0

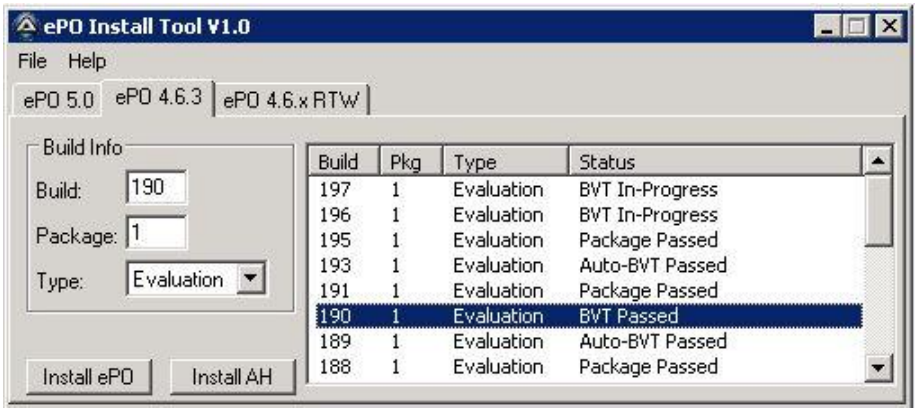


Figure 5: ePO Install Tool 4.6.3 – Version 1.0

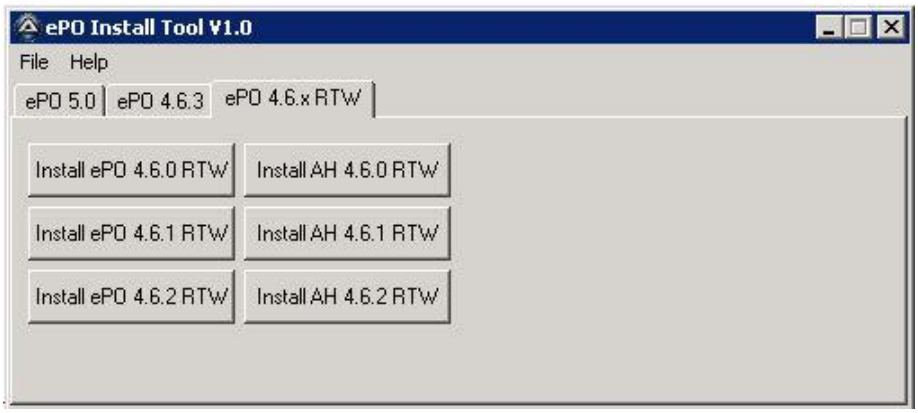


Figure 6: ePO Install Tool 4.6.x – Version 1.0

7 ePO Install Tool (Version 1.1)

7.1.1 Updates

Things were coming along pretty well and the tool was actively being used by multiple users across departments. I was still getting a lot of feedback, including:

1) Provide all of the port info for an ePO install so I can specify what I want and verify the ports aren't already in use before the install starts.
2) Provide more feedback when the install is occurring so I know what is happening.
3) Give me a way to just launch the tool and click "Install" without needing to enter all those values.
4) For debugging purposes if something goes wrong, I want to see a log file of what happened.

ePO uses a number of different ports during the install. These are all configurable in our installer, so users wanted that functionality in the tool as well. I added port info in its own section labeled "ePO Port Info" and when the tool launches, it defaults to our standard ports. The users can change these ports to anything they would like for testing purposes. I was able to find a function on the AutoIt forums that provided port checking and I used that to validate that the ports specified in the tool were not being used on the system.

Once the user clicks the "Install" button, there wasn't a lot of information displayed to the user as to what exactly was happening. Because of this, I added a status bar at the bottom of the tool that accurately displayed a status at each step of the install process after clicking "Install." When the install is completed, a dialog box is also presented to the user letting them know if the install was successful or not.

Users still wanted this process to be easier. The tool was very useful, but there were a lot of fields now that possibly had to be filled in depending on the environment that was being used. Users wanted a way to just launch the tool and click "Install." I remembered when I built the automated script for our BVT's that I used a configuration file that the script would use upon running it. I wanted to use this same concept and allow users to have a configuration file that the tool could read that would automatically input all values. All the user would need to do is select the configuration file, select the build they wanted, and click "Install." Under the File menu, I added an option of "Load Configuration File…" and "Save Configuration File…" When a user has launched the tool and input all the field values according to their environment, they would save the configuration file. The next time they used the tool, they could load their configuration file and all of the values they had saved previously would load into the tool. This would allow users to have multiple configuration files for each one of their different environments.

Sometimes the install would have a failure and users wanted to see a log file during the install so we could determine what went wrong. I added logging to the tool so that a log file was created in the same

directory as the tool and it would report the exact steps that were occurring. Based on the contents of the log file, you could determine where the install failed.

Below is a screenshot of the newly added port info and status bar.

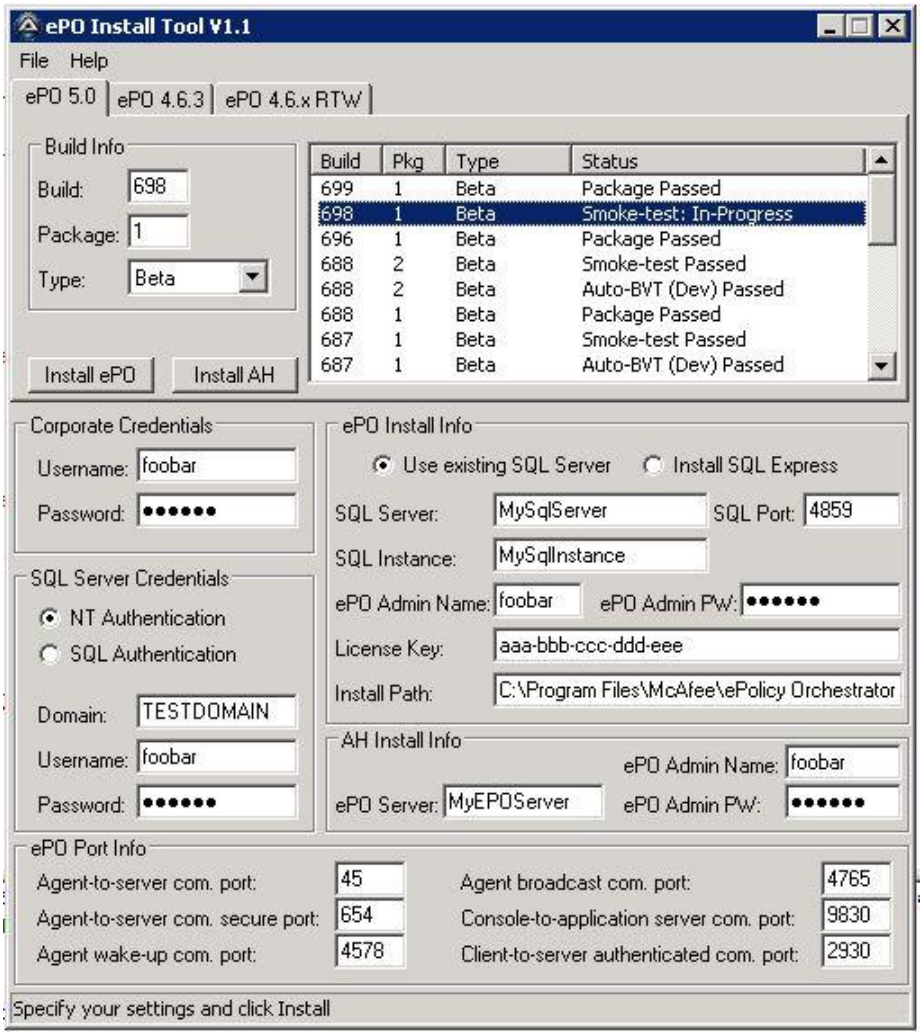


Figure 7: ePO Install Tool – Version 1.1

8 ePO Install Tool (Version 1.3)

8.1.1 Updates

At this point, the tool was very stable and was widely in use. There were approximately 20 people using the tool by this time and that number has been growing steadily. There were still things that people wanted to see added, but the majority of features had been integrated. Our Performance QA Engineer asked if he could help me work on the tool because he wanted to integrate it into his performance testing. Since I knew he had one of those CS degrees, I gladly accepted his offer to see what he could do. There wasn't a lot of UI changes that he made, but he took a lot of the logic in the code that I had written and dramatically increased the efficiency of execution.

We were able to remove the "Corporate Credentials" section because it wasn't really needed on a per user basis. Those credentials were primarily used for copying the build off of the network and we had a

general corporate account that we could use for all users. We now provide those credentials in the code instead of requiring the user to enter them.

Below is a screenshot of Version 1.3, which is currently being used.

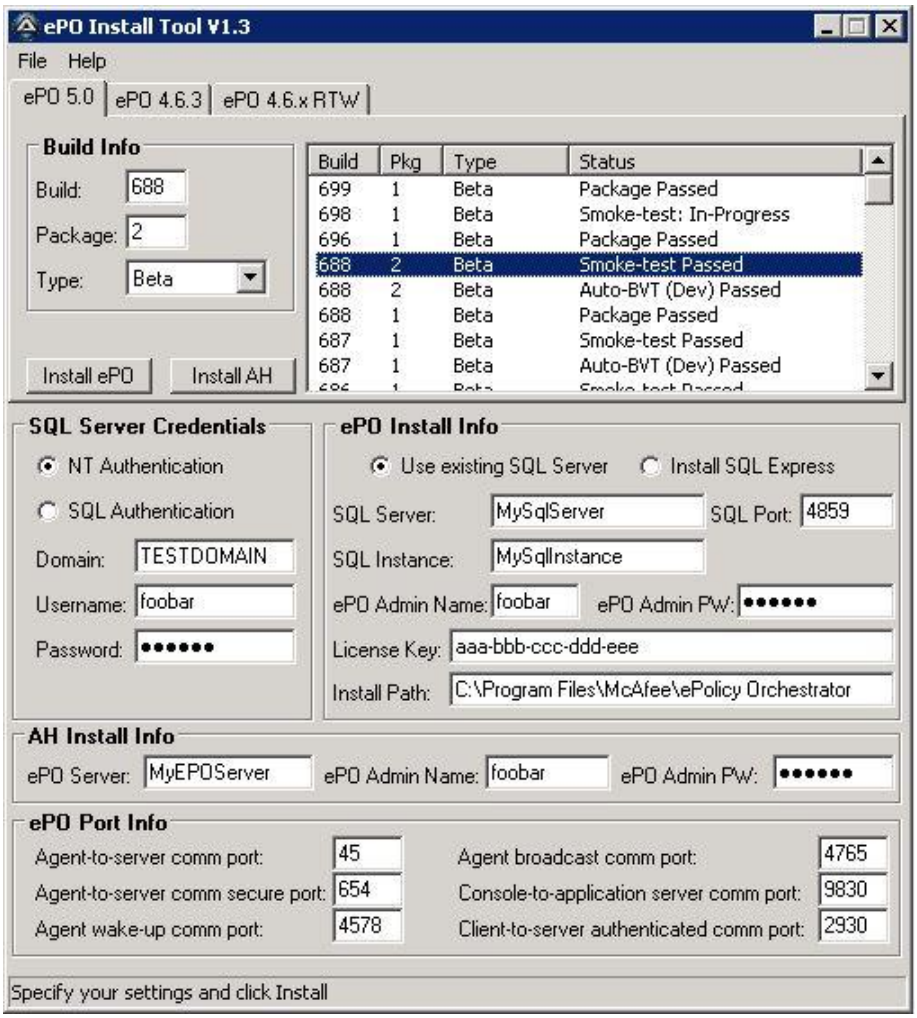


Figure 8: ePO Install Tool – Version 1.3

9 Future Updates

There are still a lot of feature requests and ideas that are coming in for the tool. It's just a matter of finding the time to get these added. Some of the features that users want to see are:

1) I want to be able to run the install on a remote system instead of only on my local system.
2) I want to use the tool to perform an upgrade instead of only a clean install.
3) I want the install to check for an existing DB on my SQL Server before installing.
4) I want more options and other product installs…

Currently the user is required to run the tool on their local system. Users would like the ability to point to a remote system and install ePO to that remote system.

The tool only supports clean installs at this time. Eventually I would like to add functionality to perform an upgrade as well.

If you attempt to install ePO using a SQL Server that already contains a DB with the same name from a previous install, the install will fail. Users are required to verify that there isn't an existing DB before the do an install. I would like to check for an existing DB before the install started. If there was an existing DB, I would prompt the user and give them the option to delete it.

I could go on and on regarding the features the users want to see. The possibilities are almost endless.

10 Summary and Conclusions

Trying to get builds installed and configured in your environment can be a daunting task, especially when a new build could be delivered as often as every few hours. Something that shouldn't take a long time can end up being a large time sync and can take away from valuable time actually performing your daily activities. AutoIt is a very powerful scripting language which can help automate some of the more mundane and repetitive tasks. It doesn't take a Computer Science degree to build a tool like this that will help increase efficiency and reduce overall cost to your organization.

References

AutoIt: http://www.autoitscript.com/site/autoit/

Occupy Cross Platform Testing

Shilpa Ranganathan

Email: shilpa.ranganathan@microsoft.com

Julio Lins

Email: julio.lins@microsoft.com

Abstract

Are you confronted with the issue of supporting an application on various platforms? Is your test matrix increasing with each release to accommodate various versions of your platform? Teams that test multi-platform products face the choice of porting tests over for each supported platform or trying to create tests that can be shared across platforms.

This paper describes our strategy and experience for platform agnostic test development across the last two releases of the Microsoft Lync product. Our proposal consists of a methodology and design for producing test content that is adaptable to application variations, allowing tests to be created once and run for multiple platforms.

Biography

Shilpa Ranganathan is a Senior Test Lead at Microsoft. She has worked in the Unified Communications space for eight years. She is passionate about working on innovative techniques to improve software testing and helping create and ship high quality products.

Julio Lins is a Senior Test Engineer at Microsoft, working on Unified Communications technologies. With a background in development of large scale corporate systems and a B.S. in Computer from UFPE in Brazil, Julio joined Microsoft 6 years ago into the Unified Communications group. Julio has been focusing on improving the test process through automation, new testing tools and techniques.

1. Introduction

Microsoft Lync is an enterprise unified communications platform. With Lync, users can keep track of their contacts' availability; send an instance message; start or join an audio-video conversation or web conference; share your desktop; or make a phone call. It replaces a table top phone with a much more complete experience across a desktop, tablet and smartphone.

This paper focuses on the challenges involved in testing the Lync client software – excluding the server side –over several releases that have been including new platforms with each new development cycle.

1.1 What is driving Lync cross-platform testing?

Microsoft Lync currently supports 13 different client platforms or variations, as well as multiple released versions. We have released a desktop application for Windows; a managed SDK API for both .Net and Silverlight that allows other companies to implement their own Lync client; a web client; and mobile clients for iPhone and Windows Phone 7.

The number of platforms we need to support is still on the rise given the market trend of platform diversification, especially in the mobile markets.

This trend drives the need for increased efficiencies around code reuse. These clients will also need to interact with each other and their multiple versions. This is the key motivator for us to investigate and invest in a strategy for cross platform testing in the Microsoft Lync team.

1.2 What is our background?

To help us identify a testable unit among the many variations of Lync clients and their platforms, we defined the concept of an endpoint, meaning a given client running on a given platform. For example, the test scenario of an instance message conversation involves two endpoints. They could be of the same type, such as two Lync desktop clients; or of different endpoint types, such as a Lync desktop client exchanging instant messages with a Lync iPhone client.

The endpoint concept may also be extended to a specific entry point to a given client. For example, the Lync desktop client being exercised through its user interfaces is one endpoint type, whereas the same client, when exercised through its public SDK, represents yet another endpoint type. This is defined as such because the implementation of the interaction with the product in each case requires different testing mechanisms.

The organization structure we follow in the Lync team has separate product teams dedicated to developing and maintaining each different Lync client. For example, the Lync Windows desktop client is developed and maintained by one team; different from the team working on the Lync client for iPhone, for instance. This goes as far as separate teams for developing and testing a specific endpoint type, such as the public SDK for the Lync desktop client.

An interesting consequence of this team division is that a given feature, say instant message, is tested by team-A for client-A, and by team-B for client-B. This is mostly due to the platform-specific knowledge required for developing and testing on each platform. In fact, each Lync client, or even endpoint type, is a separate product, with product code sharing analyzed on a case by case basis.

Features are made available on each endpoint type as the maturity of the product evolves. In the past, we chose to implement test cases and test abstractions that were client/platform specific. This gave us more flexibility and little need of coordination between feature owners as they did not have to rely on each other's tests. Given that each client had its own release schedule and set of priorities, using different test collateral gave the testers the flexibility to modify the tests as they chose fit to adapt to their schedules. Each test team implemented and maintained the same test scenarios across their respective Lync clients.

One side effect is that test scenarios were written against a test library whose signature was tightly coupled with the underlying platform layer code, making it only usable for that particular endpoint type. This meant that even the high level test steps, required for verifying scenarios such as the instance message conversation, were rewritten for each endpoint type.

Changes in the definition of common scenarios caused all the test owners to change their tests. As the product code evolved and was refactored, nearly independently for each platform, test teams had to create the test collateral from scratch for each release.

This meant that the size of each team would have to expand to accommodate this rewriting of the existing tests plus the development of new scenarios. This impacted our ability to support multiple platforms with the available resources and time. It also resulted in considerable duplicate efforts across the Lync organization.

As new Lync endpoint types are added with each release, coming up with a way to reuse the same test scenarios across different platforms and versions became a need and the possible return on this investment became much higher.

1.3 What is our strategy?

The above issues led to the following strategy: reusing a test case across different platforms needs to start during the test planning stage. If a test case is created with only one particular platform or endpoint type in mind, it will tend to make assumptions that do not apply to other platforms. Thus it may be cheaper, initially, to create a new copy of the same test case rather than to modify the platform specific one. The process that we outlined to help ensure we designed our tests to be cross-platform consists of the following steps:

Step 1: Form an organization of cross-platform feature test owners

Form a team of feature test owners of the same feature across platforms. One representative from each platform is ideal. This feature team is responsible for identifying the set of shared test cases for that feature. They are also responsible for implementing the tests and ensuring they work across platforms.

Step 2: Identify tests that should be shared

Not all tests should be shared. There are some tests that are needed to validate the intricacies of each platform; clearly these cannot be shared across platforms. For others, the ROI of sharing is not that high across platforms. Identifying a list of shared tests is the responsibility of the cross platform feature test team.

Step 3: Design and Implement shared tests

Once the sharable tests are identified and agreed upon by the cross platform feature test team, the team designs common interfaces and test code on top of those interfaces. They identify owners within themselves to implement an initial set of common tests. This helps to ensure that there is consistency in how the tests are implemented and ensures that all platform owners are aware of how the test is implemented.

1.4 Design Principles and Infrastructure to support shared tests

Our goal was to create a unified set of tests, test libraries and a framework, so that the same set of tests can target all Lync platforms. The high level strategy that we adopted was:

a) Create a framework to expose interfaces that will be based on the scenarios that a user can accomplish using a Lync client, such as, “Make a call”.

b) Create tests that use the functionality exposed by the “interoperability interfaces” (from step a) rather than use the underlying product testing mechanism directly (which is endpoint specific).

c) For each different endpoint type, there may be an adaptor layer necessary to translate from the “interoperability interfaces” to the actual endpoint implementation in case the interoperability interface is incompatible with the underlying object model.

1.5 Verification mechanisms

Sharing test code requires a conscious effort, as it demands constant policing to avoid duplication of code, or, even worse, common code bound to a specific platform. Hence we put the following verification mechanisms in place to ensure that the teams were following the process and design principles outlined.

Code reviews of the shared tests**,** interoperability interfaces and adaptors were performed by the cross platform team to ensure the tests meet requirements for all platforms. They validate if shared test cases are following the identified design principles, hence avoiding any interaction with platform-specific components or APIs (object models).

Shared test cases are also not allowed to make any assumptions of endpoint specific behaviors, such as types of users, platform-specific feature behavior (unless abstracted by the generic interface), phone numbers, etc.

Each shared test has a set of tags to indicate which platforms they have been verified to have passed for. Not all product versions across platforms might have the exact same feature set. Even though a test case is multi-platform, it does not mean that the case will run for every platform. The test case shall run for the platforms which the test scenario is valid for.

Test owners have the responsibility to identify which tests are run on which platforms and will tag the tests accordingly once they have validated that test works for their platform of responsibility. This helps when the test needs to be modified. Once modified the test should be re-validated on all supported platforms, as indicated by the tags, before the check-in.

2. Implementation overview

The main purpose of the shared code design is facilitating the reuse of a set of common scenario interfaces across different products, while keeping the code isolated from platform particularities.

Microsoft Lync testing involves a vastly different set of platforms, among them Windows desktop, Windows RT, iOS, Android and Windows Phone. Since not all those platforms have considerable commonalities, we chose an approach that allows us to run test cases on the desktop and interact with the product applications running on remote devices.

While it could be feasible to write the scenario driver portion on the device itself, it would require a common language (perhaps plain C++; or a script language) and platform adaptors so that the test case could be triggered (run) on all platforms.

So the main advantage of running test cases from the desktop is that the test cases can be written in C# and run on a Windows desktop, making full use of the .Net platform, while each endpoint wrapper is written specifically for each platform, hence making full use of its host platform. What makes the communication possible between the test case and an endpoint is a communication channel; which has been implemented by either .Net Remoting; or a simple in-house XML based protocol for remote procedure call in test cases.

Another logical reason for running the test case outside of the device is that most Lync scenarios involve multiple endpoints. For example, exchanging an instant message involves two users, each signed into

their own device. So the support necessary to coordinate multiple endpoints would have to have been implemented for all platforms.

Before exploring the in-depth design of our test framework, it is necessary to establish a set of requirements imposed by multi-platform testing.

2.1 Multi-platform testing requirements

Our strategy of multi-platform testing focuses on sharing reusable units across the different Lync endpoints. The items listed below constitute reusable units that we intended to share:

- **Full test cases** that verify common scenarios across platforms. Examples: send and receive an instant message, make an audio call, change availability, etc.
- **Partial scenarios** that are used by shared test cases. Examples: send an instant message, receive an instant message, start an audio call, accept an incoming audio call, etc.

Thus, the requirements are intended to make it possible to reuse both full and partial test scenarios across multiple products. A few requirements should thus be enforced:

Test cases and partial scenarios are isolated from implementation details: each test should be agnostic of implementation details or APIs specific to a given platform. This can be enforced by the namespaces that the test case code is allowed to access.

Configuration details are external to the test case code: test setup or configurations pertaining to specific platforms should be implemented outside of the scope of the test case code.

3. Framework elements

The design of the framework has five main elements, listed below and shown on Figures 1 and 2.

Single-Client Library (SCL): defines a set of interfaces that are generic enough to represent any Lync client object model. The interfaces are implemented for each client on top of their Test APIs through an adaptor or directly implemented by the test APIs.

Multi-client library (MCL): implements scenarios on top of the SCL that involve multiple endpoints, for example, establishing an instant message conversation between clients A and B. Since the scenarios are implemented on top of the SCL, the MCL can be used on any client platform.

Test API: implements an encapsulation of the actual product API or GUI that is exposed for testing. Usually abstracts some of the complexity of the underlying model, such as an asynchronous API or UI automation framework. The Test API implements the SCL interfaces directly or through an adapter.

Endpoint manager: allows a test case to create the required endpoints without knowing about implementation details of those endpoints. All the information necessary to connect the SCL interfaces with the implementation is metadata external to the test case.

Service interfaces (SI): represents concepts such as IFileSystem (exposing a generic interface to interact with file systems that could be different for different platforms) or IRegistry (exposing a generic interface to interact with registry) that offer a standard way for test cases to execute their setup requirements. These services may or may not be available for a given platform.

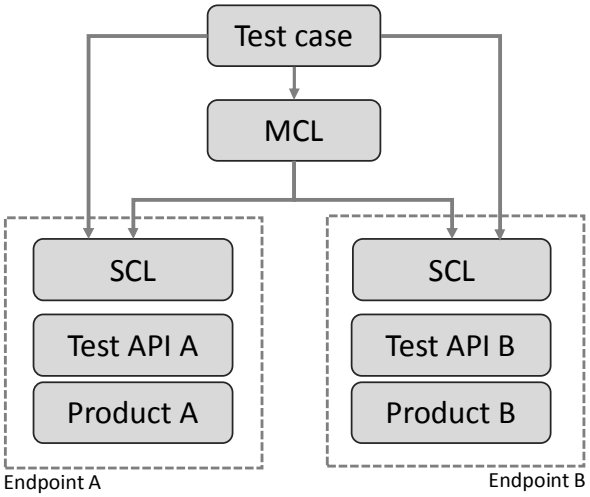


Figure 1 – relations between Test case, MCL and SCL

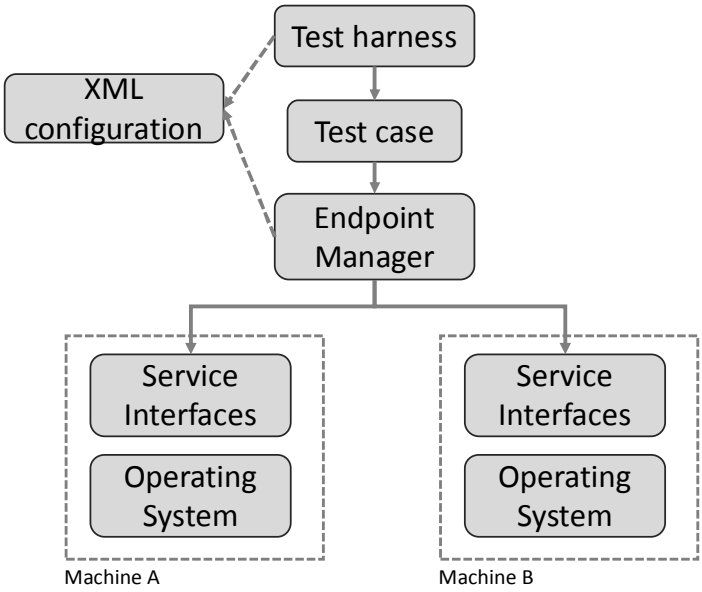


Figure 2 – relations between the Test Case, Endpoint Manager and the Service Interfaces

Figure 1 depicts the relationship between the test case and the two components that build up complete reusable scenarios: the SCL and the MCL. The SCL is used directly by the test case when it needs to make atomic actions or verifications that involve a single endpoint, such as Sign-In. The MCL is used for composing actions into a small or complete scenario.

The MCL implements a set of highly reusable scenarios implemented on top of the SCL, hence they are independent of endpoint type. The MCL provides a way for a test case to reuse at least part of the scenario of a similar test case.

A typical use of the MCL involves interacting with multiple endpoints to achieve a scenario. For example, to establish an audio call, it is necessary make a call from endpoint A, accept on endpoint B and verify each endpoint is connected after the call was accepted. So, one can visualize the MCL as an aggregation of SCL actions and verifications.

A test case may use one or more MCL methods to achieve the scenario it needs to verify. The MCL should contain whatever part of test cases that have a good chance of being reused by other test cases. For example, consider the test cases Establish-Audio-Call and Hold-Retrieve-Audio-Call. Both test cases need to establish an audio call. Moving the establish call scenario to the MCL allows more complex test cases to reuse the same test steps for any endpoint type.

Figure 2 shows the role of the endpoint manager and the service interfaces. The test case requests an endpoint to the endpoint manager which, in turn, uses the service interfaces hosted on a given machine to create the endpoint. The number of endpoints spawned on each machine depends on constraints imposed by the endpoint type. For example, a mobile device will usually allow only one endpoint per device.

The actual endpoint type to be used for the scenario execution is specified in the XML configuration. The configuration will contain all the necessary dependencies for the test run, such as device types, number of devices, binaries, supporting files, all of which are consumed by the test harness in the process of automatically setting up the test execution environment.

The configuration also contains the definition of the endpoint type to be used. The definition is essentially a pointer to an endpoint factory which knows how to start the endpoint using the services offered by the Service Interfaces. The Endpoint Manager loads the factory as a plugin and delegates to it the task of creating and disposing endpoints.

The Endpoint Manager allows test cases to request the creation and disposal of endpoints without actually knowing the type of the endpoint being used. This allows the same Establish-Audio-Call test case to be used for two Lync Desktop clients as well as for two Lync Windows-8-Style clients (former Metro style clients).

The test case may also need to use the service interfaces to achieve its scenario, such as disconnecting the endpoint network while a user is signed-in. Such use of the service interfaces directly from the test case will limit the endpoint types which the test case is compatible with, since not all platforms will offer the same services. Some tests do directly use service interfaces, despite this limitation.

4. Lessons learned

While reusing tests across platforms increases the return on investment as we support additional platforms, supporting cross-platform tests increases the cost of test development. In this section we describe pitfalls, lessons learned and some best practices we put in place to help mitigate some of the pain points.

4.1 Standardization of APIs across products

The similarity of the product APIs across platforms has a direct impact on the amount of effort necessary to establish a common set of shared tests. Products that expose a standard API for testing across platforms offer a higher return of investment for sharing tests.

It is important to note that if the product itself follows a layered architecture and shares code among the platforms on which is runs, it will more likely expose similar APIs across platforms, hence it will positively impact on the reuse of test code.

When the product APIs are similar between platforms, it will be easier to define common interfaces for the same functionality across platforms. Whereas, when the differences are great, adaptors might be needed for connecting between very different APIs. Another drawback of diverse product APIs is that arriving at a common set of test interfaces will require more effort from the feature test team in their pursuit of agreement on what the common code should look like.

For future releases we are investigating the feasibility of standardizing the application layer interfaces for the different client types, in an attempt to have a similar API set exposed for testing for all clients.

4.2 The engineering infrastructure impact

The engineering aspects of the company greatly impacted the feasibility of sharing test code.

Our experience in testing Lync indicates that it will be more productive to share test scenarios among products that share the same source code management system. In recent development cycles, our product source code was distributed among three source code management systems. The end result was that products sharing the same source code management systems would live up to its full test sharing potential, whereas products on different source code management systems would have partial sharing, dependent on other variables as follows.

The build process and tools influenced the cost of sharing. In the ideal scenario, all products and test code are built from the same source control system and tests are executed upon successful builds.

Breaks are detected as soon as they are introduced. In circumstances where the product and its tests are built and maintained by separate teams in separate source control systems, breaks in functionality that were introduced by one team will take longer to be detected and hence impact the productivity of all teams involved.

Finally, the ability of triggering test runs for all platforms impacted the sharing process. In the ideal scenario, a tester of a given platform is able to kick off test runs of the same scenario for other platforms with minimal test setup efforts. This allows changes to be verified across all target platforms. The more complicated the test setup for a given platform is, the greater will be the cost of validating changes for that platform. This will ultimately impact on the team's ability to verify product changes when product code is shared across platforms.

We are pursuing conversations with the various teams to try and consolidate the source code management systems for the different clients. While some teams are agreeing to this we will still have some teams using different source code systems at least for the next release and we will continue to pursue this discussion with them.

4.3 Schedule

When different teams are working on each platform, differences in schedules and priorities also greatly contribute towards the success or failure of sharing tests. Even though teams may intend to share tests, one team may be further along in the product development cycle, and hence may have a better idea of the feature set to implement and test. In order to conform to their schedule needs, they may have to go ahead in developing the tests without much knowledge of the requirements from other teams who are not as far along in the product development cycle. This may lead to developing tests that might not fit the need of all platforms. In our experience, teams with a stable test base may be averse to considerable refactorings, or any at all, whereas teams joining a common test code base will demand adjustments in interfaces or partial scenarios when they differ much from their product behavior. This conflict of interest in the common code base may result in new teams only partially adopting the shared tests. Hence, differences in schedules may adversely impact the sharing process, possibly reducing its scope.

For example, the Lync desktop client was the first to introduce the feature to share the desktop. At the time this feature was introduced, no other client was ready to implement this feature, so the test owner designed and implemented the common interfaces and tests based on the feature requirements for the desktop client. We had around 20 tests stabilized and running successfully, when the decision was made to add this same feature in the managed client platform SDK. We found that modifications were necessary to the existing tests, as the way to initiate desktop sharing between the 2 clients was considerably different. One client exposed a single API to initiate the sharing, whereas on the other client the initiation was split into 2 API calls. This basic change impacted all the tests. Clearly the desktop client team had reservations around changes to these tests as they now had to modify all tests as well as validate all the tests to ensure no regressions. The outcome of this conflict was that we introduced a new interface to expose the sharing initiation. This change was made by the managed client platform SDK owner and then both teams had to execute the tests based on these changes against their platforms. After consecutive bug fixes, we were finally able to stabilize the tests for both platforms.

Schedules are based on business and market trends so we will continue to face this problem.

4.4 When full tests cannot be shared, share smaller scenarios

It was our experience that not all test scenarios were exactly identical among platform specific versions of our product. The feature set is not an exact match across the board. Features may be restricted to a given platform set of capabilities. This is also aggravated when the underlying product object model differs between platforms.

While each platform may have a specific set of features, there are likely to be common scenarios that are shared across platforms. For example, all Lync clients share the Sign-In and Instant Message features, but, at the time of this writing, not all clients offer the VoIP call feature.

One interesting aspect of the approach we took is that the common interfaces (SCL) have a generic reference (object type in C#) to their underlying implementation. Although this may seem odd from a common interface design, it allows test cases to access the underlying implementation, hence its platform specific actions, when needed. The use of an object type property instead of casting is intended to enforce the support to the cut-through concept.

This is useful for platform specific scenarios that start from a common scenario. It allows a platform specific test case to reuse the code necessary for achieving the common scenario, then cut through the interface layer to access the platform specific code. For example, the steps involved in testing a VoIP call include signing-in and creating a conversation, both of which are scenarios that can be reused across all platforms.

It becomes essential to not only identify reusable scenarios, but to also define which parts of these scenarios will be platform specific and which parts can be reused across platforms. Hence, the role of the cross-platform feature team is not just defining the ideal reusable scenarios, but also focusing on the most return of investment for tests shared across platforms.

4.5 Latent costs associated with sharing tests across platforms

The reuse of tests adds additional costs to the automation development. Although we are unable to provide a formula for the return of investment in test reuse, this section describes some situations in which we observed additional test development costs.

Differences in development environments will delay the development and verification of common test code. Sometimes one test owner does not have the specific environment to build/test on a particular platform. In this case he/she has to wait for that environment to be setup or ask someone else to run the test on other platforms.

The coordination among the feature teams across platforms to define the shared test cases and create a test design to accommodate all platforms adds more time to the process. For example, the code review process for a change in common code may take longer as all stake holders, in our case from different teams, need to review and approve it.

The stabilization of the tests on the various platforms is certainly time-consuming. Although each test scenario is not replicated for each platform, it still needs to be stabilized on the target platforms. We found that once a test has been stabilized on one platform it takes approximately one forth the amount of time to stabilize on new platforms.

We are working closely with our leadership team to ensure that these costs are accounted for during the planning stage so that we are better prepared.

5. Conclusion

Reusing tests is not exactly a straight-forward process. While one may likely provide very strong arguments for the reuse of product code, test automation code is *one of* the tools for asserting the product quality. The reuse of test is a trade-off between the cost of development and the final impact on the product quality, often measured by the amount of bugs in product code found by automation.

The amount of code to be shared will be impacted by the variables mentioned in the previous section, involving the engineering aspects, such as source code location, as well as organizational aspects, such as team schedules and the disparity of the product characteristics across platforms.

As the title of this article jokes with, cross platform testing will very likely bring up special cases with impracticable reuse costs. A wise test organization will look to trade-off the amount of tests reused among platforms, balancing between the savings of reusing code with the need for platform-specific testing, given by platform-specific features or prohibitive reuse costs.

This article presented some guidelines and a framework to be carefully analyzed by a test team working on a multi-platform product, and intending to save on test development costs by reusing test scenarios.

In the Microsoft Lync team, we are continuing to expand our investments in the engineering infrastructure to create a common framework to support all platforms. Even though we are unable to measure precisely the return of investment in reusing test code, our experience indicates this is a vastly improved process and that the Lync test team has embraced it. Having seen it at work, we can't imagine going back to a world where we replicate all test efforts for each new platform.

State-based testing using dynamic instrumentation – A Case Study

Author
Vijay Upadya, vijayu@microsoft.com

Abstract

Often times while testing state-based systems, testers have the need to validate the behavior of software on performing a specific action while a component is in a specific state. However this is not easily possible using traditional testing techniques because the state transitions are either not visible/observable externally to tests or, in cases where they are visible, are not controllable in real time by tests. One technique that addresses this challenge is dynamic instrumentation, in which the system under test is dynamically instrumented using binary interception techniques that then enable tests to observe and control the product as it transitions through its various states.

Testing state-based systems can be quite challenging and costly. The primary challenge in testing such systems is the need for deterministically covering all the states and state transitions of interest in an effective and efficient manner. This paper will talk about the technique we are using in our team to test such a system using dynamic instrumentation. The paper will discuss this testing approach, the tools used to do dynamic instrumentation and how to write tests to leverage the dynamically instrumented product code to simulate different states and state transitions. This paper will also demonstrate how the approach lends itself well for creating data driven tests and ways it can be used to augment the existing test automation.

Biography

Vijay Upadya is a senior software design engineer in test at Microsoft's Redmond campus in Washington. He has 14 years of industry experience and is currently working in the Microsoft Windows Live group and primarily focuses on test strategy, test tools development and test process improvements for the team.

Vijay Upadya has Master's degree in Systems Analysis and Computer Applications from the University of Mangalore, India and Bachelor's degree in Mechanical Engineering from Gulbarga University, India.

1 Introduction

Testers testing state-based systems have a need to validate the behavior of the system under test (SUT) on performing a specific action while the product is in a specific state. However such testing is not easily possible using traditional test approach because:

- The state transitions of the SUT are either not visible/observable externally or in cases where they are visible, they are not controllable in real time.
- Traditional functional tests run out-of-process with the system under test.

Such was the situation when our team took on testing file synchronization software (Wikipedia 2012a) named *SkyDrive* (Wikipedia 2012b). This software is a client application that synchronizes files across users' devices as well as synchronizes files to their stored 'cloud' locations. Although the client application is a state-based system, not all of its states and transitions are externally visible to the end user. The behavior of the application varies depending on its state, which is changed through user actions. For example, the code path followed by the application's execution is different while the user is editing a file than the code path followed while that same file is being uploaded. And the code path is yet further different when the user is editing a file while the application is in an idle state. Trying to test these states and transitions posed a great challenge to the team.

2 Approach

The key to being able to test different conditions was to be able to observe and control the state transitions in the product. The solution the team used to achieve this level of control was *dynamic instrumentation* of the SUT. In this approach, key functions in the product that cause state transitions were identified. Instrumentation code was then added to notify the test program when different state transitions happened. The test program then performed appropriate steps to trigger the state transitions and was able to validate the SUT behavior deterministically (i.e., in a repeatable fashion).

The framework that we used for this dynamic instrumentation was *detours*. *Detours* (Galen, 1999) is a library for instrumenting arbitrary Win32 functions in Windows-compatible processors. The *detours* framework provides methods to intercept Win32 functions by re-writing the in-memory code for target functions. The library also contains utilities to attach arbitrary DLLs and data segments (called payloads) to any Win32 binary. One of the big advantages of this approach is that since the interception happens at the binary level dynamically, instrumentation doesn't require any change to the product source code. *Detours* express library can be freely downloaded from the link given in the reference section at the end of this paper (Microsoft Research, 2012).

We used *detours* functionality to implement a test library, which could then be used to observe and control the SUT in real time as it moved through its various states. In its supported usage, a test library implemented with *detours* can be injected (i.e., loaded) into the application process dynamically at run time while executing the tests.

3 Implementation

3.1 Basic setup

- The test program (Test.exe) used *DLL injection* (Wikipedia 2012c), which is a technique used to run code within the address space of another process by forcing it to load a dynamic-link library (i.e., a DLL). The *detours* test DLL (detourTest.dll) is injected into the product process (Product.exe) explicitly by the test and detour methods that are appropriate for the test are enabled.
- The *detours* test DLL *detours* necessary product functions (could be public or private). Since the *detours* DLL runs in-process with the product process, it has full access to the internal state of the product.

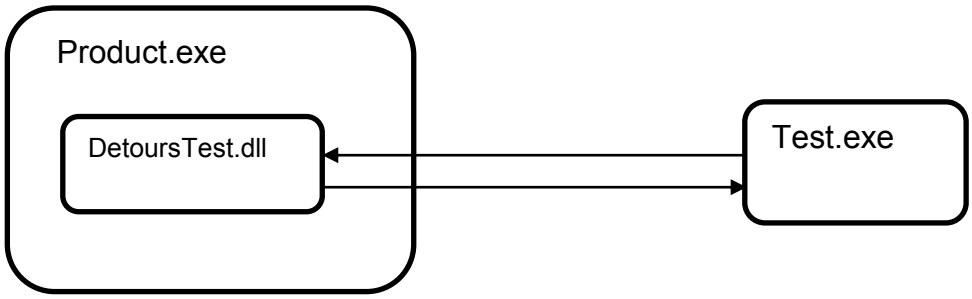


Figure 1: Test execution using detours

In the above diagram, the Test.exe launches the system under test (Product.exe) and loads the DetoursTest.dll into the product process explicitly. Then it sends commands to the SUT to trigger various state transitions and waits for event notification back from DetoursTest.dll.

3.2 Test Steps

- The *detours* component fires events for all product state transitions that are instrumented for a given scenario
- The test program records various events fired for that scenario
- The test program then inspects these recorded events, repeats the scenario and performs appropriate actions while product is in a specific state. For example, test.exe can perform a file edit while that file is being uploaded.
- The *detours* DLL also enable tests to 'block' the product when it reaches a specific state to give the test program an opportunity to perform its actions deterministically (else by the time an action (like edit file) is performed, product might leave that state and transition to a different state).

All of the above steps are automated and don’t require any user interaction.

The following example will demonstrate this sequence more clearly:

3.3 Example

Let's take an example test case of editing a file while the product is uploading it.

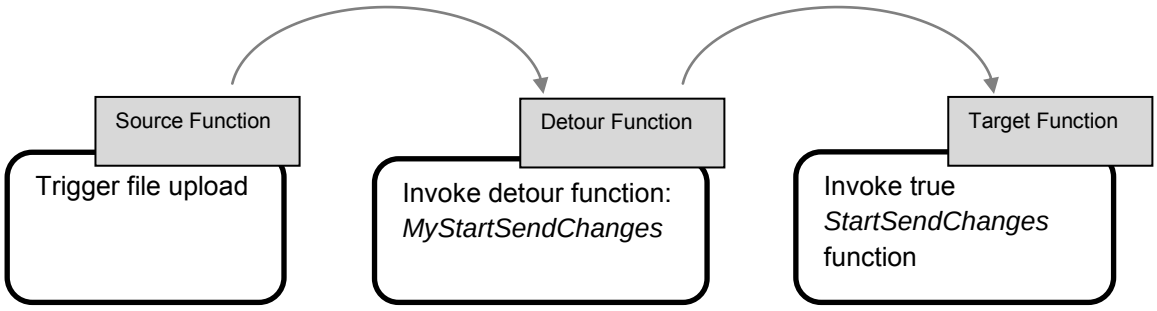


Figure 2: Detours execution flow

Step1: Identify the method in the product that performs file upload. Say this is the signature of that method -

```
void StartSendChanges(Changeset* changes)
```

Step2: Create a test wrapper in DetoursTest.dll for the product function 'StartSendChanges', which overrides the product code by using the *detours* method - DetoursAttach. When the product code calls 'StartSendChanges', the actual function being executed will be 'MyStartSendChanges', allowing Test.exe to record the product state."

```
void MyStartSendChanges(Changeset* changes)
{
	SignalAndWait(ProductStates::UploadBegin);

	//Call real product function
	StartSendChanges(changes);
}

//Attach the detour method
DetourAttach(StartSendChanges, MyStartSendChanges);
```

DetourAttach is a helper method that is part of detours library that redirects the product's function pointer to wrapper functions. *SignalAndWait* is a helper function called from inside the detour wrapper method *MyStartSendChanges* shown above. This function fires '*UploadBegin'* event to notify listeners (test code) that the product is in the 'Upload' state and waits until it is signaled by the test. Once signaled, the original product function *StartSendChanges* is called to resume the execution flow. Figure 2 above shows the execution flow.

Step3: In the test following steps are performed

- Start product
- Inject *detoursTest.*dll to product.exe
- Trigger file upload
- Block the product when it goes to 'upload' state
- Perform file edit. At this point, the product is guaranteed to be in 'upload' state
- Resume upload
- Verify that upload resumed and completed successfully

4 Data driven tests

Once the basic flow of the test was identified and implemented, the next task was to prepare the list of all possible state transitions that needed to be tested and actions that needed to be performed in each of those states. The test team identified about 30 states (e.g., file upload begin state, file upload complete state, etc.) and 15 user actions (e.g., file edit, file delete, etc.) that needed to be tested. In order to test all these cases efficiently, we built a simple data driven test (Wikipedia 2012d) framework that took a list of actions and states in the form of an xml document and generated test cases by combining these two pieces of data.

For example, let us say we have to test four actions-: edit, delete, move and rename of files while the product is a) uploading and b) downloading the file. This information is specified in an xml file as shown below:

```
<TestData>
  <Actions>
    <EditFile/>
    <RenameFile/>
    <MoveFile/>
    <DeleteFile/>
  </Actions>
  <States>
    <Uploading/>
    <Downloading/>
  </States>
</TestData>
```

The data driven test framework parsed the above xml data and generated 8 test cases (4 actions x 2 states). Each of the test cases was then executed as explained in the above section 3.2. This data driven test approach enabled the test team to rapidly test in an efficient manner.

5 Coverage and Cost

Before we adopted the state-based testing approach, more than 60% of states in the product were just not possible to test deterministically because there was no way for the test program to observe and control these state transitions. The test team had no test coverage for these areas. This state-based testing approach enabled the test team to rapidly test these missed states and get the much needed test coverage.

In terms of the investment, it took 15 person days to get the *detours* based test infrastructure up and running. Then it was just a matter of incurring incremental cost for defining new states and actions and enabling them in the *detours* test library. Due to the data driven nature of test definition and execution, this incremental cost is much less than if the same tests were to be written in a non-data driven way. For example, a typical test would take approximately half a day to one day to implement, whereas a data driven test would typically take about an hour to implement.

6 Other Applications of Dynamic Instrumentation

While the above section shows how *detours* can be leveraged to do targeted functional test, there are other applications of dynamic instrumentation. Below are some of the areas that can be applied:

- **Deterministic/Controlled stress testing:** Perform various real world operations (edit file, delete file, network off, etc.) while file synchronization is progressing through various states.
- **Crash resiliency testing:** Crash client while it is in a specific state and test its resiliency (restart and sync should continue from where it left before crash)
- **Delay/Timing resiliency testing:** Run existing functional tests by introducing random delays in various stages of syncing process. This is to catch any timing related issues in the product.
- **Error testing:** Modify return value of function calls to simulate error conditions. This is a cheap way to test error conditions that typically tend to be expensive/hard.

7 Conclusion

Dynamic instrumentation using *detours* enabled testing of state-based systems effectively and improved the coverage of testing different states. This methodology does require initial investment to identify different states and instrument the product code based on these states, but once it is in place, it can rapidly enable test teams to test different conditions/code paths in the product in a deterministic manner.

Using this framework, and our data drive test methods we could cover 60% of additional state transitions that were not possible to test before and add new tests in an hour instead of a day.

8 References

Wikipedia 2012a, File Synchronization, http://en.wikipedia.org/wiki/File_synchronization (accessed August 16, 2012)

Wikipedia 2012b, SkyDrive, http://en.wikipedia.org/wiki/SkyDrive (accessed August 16, 2012)

Wikipedia 2012c, DLL Injection, http://en.wikipedia.org/wiki/DLL_injection (accessed August 16, 2012)

Wikipedia 2012d, Data-driven testing, http://en.wikipedia.org/wiki/Data-driven_testing (accessed August 16, 2012)

Hunt, Galen and Brubacher, Doug. 1999. *"Detours: Binary Interception of Win32 Functions".* *http://research.microsoft.com/apps/pubs/default.aspx?id=68568*

Detours, Microsoft Research, 2012. http://research.microsoft.com/en-us/projects/detours/

30 Years of Regression Testing: Past, Present and Future

Jean Hartmann
Test Architect
Office Shared Services - Engineering
Microsoft Corp.
Redmond, WA 98052
Tel: 425 705 9062
Email: jeanhar@microsoft.com

Abstract

To coincide with the 30th anniversary of PNSQC, my intent was to provide a chronology of one of software testing's least glamorous and yet vital activities – regression testing! This short paper describes the evolution of regression testing strategies through the decades, starting out as a research topic in the early 1980's, followed by development and early deployment by industry R&D labs in the 1990's to adoption by software companies in the 21st century. Each decade is described in terms of the challenges faced by regression testing, projects are highlighted in which these strategies are applied and the future of these techniques is explored.

Biography

Jean Hartmann is a Principal Test Architect in Microsoft's Office Division with previous experience as Test Architect for Internet Explorer and Developer Division. His main responsibility includes driving the concept of software quality throughout the product development lifecycle. He spent twelve years at Siemens Corporate Research as Manager for Software Quality, having earned a Ph.D. in Computer Science in 1993, while researching the topic of selective regression test strategies.

1. Introduction

Regression testing is an unsung hero of the software testing world – this indispensable testing strategy is used daily by many software companies, worldwide. Wikipedia defines regression testing as: "*Software testing that seeks to uncover new software bugs or regressions in existing functional and non-functional areas of a system after changes, such as enhancements, patches or configuration changes, have been made to them. The intent of regression testing is to ensure that a change, such as a bug fix, did not introduce new faults. One of the main reasons for regression testing is to determine whether a change in one part of the software affects other parts of the software*".

The topic was first discussed in papers presented in June 1972 at the inaugural software testing conference in North Carolina, organized by Bill Hetzel [1]. References to regression testing can then be found in the classic software testing texts, such as 'The Art of Software Testing' [2].

At the time, the focus of regression testing was on validating mainframe software. With limited computing resources and few, if any, test automation tools available, most testing was manual. Regression test passes often took weeks to complete.

Towards the late 1970's, researchers started to look at ways to reduce these lengthy, arduous and error-prone regression testing runs. Instead of executing all test cases every time one or more code changes were applied, known today as the *retest-all* approach [3], why not only run a subset of tests? The phrases regression test selection, test minimization or test prioritization were coined [4]. Thirty years later, researchers from academia and industry continue to work on the topic.

The goal of these regression test selection strategies remains the same:

- Determine a subset of tests for less cost than rerunning all tests [3]
- Ensure that this subset is 'safe' aka can catch product bugs compared to retest-all

The remainder of this paper chronicles the story of regression testing and in particular, regression test selection since 1980. It highlights key advances made by both academia and industry in developing efficient and effective regression test selection techniques. It demonstrates the time taken for techniques to transition from theory into practice. Sections 2 through 4 discuss how the technique has evolved through the decades. In Section 5, we examine how it is influencing testing today and discussing its future role.

2. 1980s: Genesis

In the early 1980's, Fischer [5,6] was the first to explore the topic of regression test selection. He applied mathematics in the form of operations research to model the problem, which is known as a set-covering problem [7]. The application of linear or integer programming techniques allowed him to identify a subset of test cases that provided coverage of modified program code, ensuring that each modified code entity was exercised by at least one test case. The model he used consisted of:

- **Objective function** that represented each test case and its associated cost
- **Set of constraints** that reflected each test case that executed a code entity

The set-covering problem is known to be NP-hard, meaning that in theory, it cannot be solved in polynomial (aka a reasonable amount of) time. In practice, therefore, there was no guarantee that the model solver (tool) Fischer used could determine an optimal solution to any given problem. Computing resources at the time were limited and so applying his strategy to large-scale problems was not feasible.

This minimal solution to selecting a subset of tests was also not deemed 'safe' by the testing research community at large. They preferred to explore other approaches that picked more tests for rerun in order to raise the likelihood of finding the same number of bugs compared to the retest-all approach. With regards to picking a safe or more conservative set of tests, Fischer's work represented the riskiest of the different approaches. Nevertheless, his work represented a milestone in regression testing research and would be vindicated in later years.

In the mid to late 1980s, when software engineering, and especially software testing, established itself as an important part of the software development lifecycle, there was renewed interest in regression testing. Researchers, who had developed new testing strategies, such as data-flow based testing [8], combined them with impact or ripple effect analysis techniques to select tests based on those changes [9,10].

However, these strategies had serious limitations in practice. They could not easily scale to the testing of large code bases. Dataflow-based techniques, which were used by many researchers, required the computation and analysis of so-called *def-use* pairs prior to test execution [11]. Each pair represented the definition and use of a program variable. Computing such pairs for large, real-world code bases required the use of complex code analysis techniques that were expensive in terms of compute time. These techniques would become even more complex when analyzing code containing numerous code functions [12] and handling language constructs, such as pointer variables, which were being introduced with the C programming language at the time [13]. These analysis techniques were known as intra- and inter-procedural analysis. In addition, the majority of the dataflow-based regression test selection techniques did not necessarily derive a minimal subset of tests.

Meanwhile, in industry, the majority of applications were now being developed in C. Code was increasing in size and complexity. The introduction of the first test automation

tools enabled companies to move from manual to automated testing scenarios to improve test reliability. Automated regression test suites began to grow in size, raising the need for efficient and effective regression test selection strategies.

In summary, the 1980's saw the genesis of regression testing research. While researchers could develop theoretical approaches, they could only explore their practical application by means of toy examples. Any tools too that were developed to support the proposed techniques were very limited in their analysis capabilities, especially when faced with the newer and more complex programming language constructs being introduced.

In the next decade, these techniques become more mature – existing approaches were refined, more industrial-strength tools and test suites were built to validate techniques. The 1990s became the decade of empirical studies for regression testing.

3. 1990s: Putting Theory into Practice

The 1990's saw regression test selection theory being put into practice. Researchers, realizing the high cost of developing tools to adequately support their investigations, turned to industry for support. They started collaborations with the large industrial research laboratories such as Bell Labs [14] Bellcore [15] and Siemens [16] to develop tools to automate the collection and analysis of the required data. Some academic research centers pursued similar initiatives, but at a lower cost and with fewer capabilities. They typically built tools on top of public domain compilers such as the GNU compiler resulting in toolsets, such as Combat [17].

Using these various tools, the teams launched extensive empirical studies in which larger pieces of code, typically C code from the public domain, were targeted for analysis and test suites were developed to exercise and revalidate them [18]. Such studies typically seeded faults in the program code, deemed them adverse program modifications, and then applied the respective revalidation strategy to demonstrate the ability to select a subset of tests and safety of that subset with respect to fault detection.

A major factor enabling the development of such tools was the availability of powerful, yet affordable, personal workstations such as those developed by Sun Microsystems. Running the required code analyses, which a decade earlier had practical limitations in terms of scalability and performance on early PCs, were now possible. Both academia and industry also leveraged this computing power to address the complexities of analyzing a new programming language - C++. Object-orientation, with concepts such as inheritance and templates, introduced additional challenges for regression testing researchers. They needed to adapt their established techniques and tools [19].

During the early 1990's, researchers began to revisit Fischer's approach, applying algorithms that could now overcome the theoretical limitations of NP-hard problems [20,21]. These so-called greedy algorithms represented heuristics that could provide a

solution quicker than model solvers, but their solutions were not necessarily minimal or optimal.

By the mid-1990s, Siemens researchers, encouraged by the results of these empirical studies and advances in tools, encouraged their industrial business units to explore the potential benefits of regression test selection. One Siemens division that researchers believed stood to benefit from this regression testing technology was the telecommunications division, which developed large public telephone switches (including the required software) for the US and world markets.

Their system test passes, by now largely automated, took approximately six weeks to complete. However, during that time, code fixes or updates/patches were made to the switch software on a regular basis, but the test pass could not be stopped and restarted, because of time and cost constraints. Therefore, the team needed a strategy and tools that could quickly identify the impact of code changes and determine a minimal subset of system tests for those changes. Siemens researchers responded by building a custom toolset based on Fischer's work that identified a subset of tests for rerun and kept the test pass within schedule. It was an early example of where regression test selection was to be used over the next decade – optimizing *post-checkin validation* or system testing processes where large, automated test runs needed to be streamlined.

One distinct twist in this regression testing evolution was its application to hardware or IC testing. Test suites in the hardware world, or so-called testbenches, typically contain large numbers of test cases or test vectors, for validating large ASICs or custom ICs. Companies, such as Intel, are always looking for ways to make chip testing more efficient. In the late 1990's, a small, U.K. startup company called TransEDA, which was developing code coverage analysis tools for Verilog and VHDL, incorporated regression test selection strategies into their VN-Optimize tool [22]. Based on this technology, hardware companies were able to prioritize or reduce the number of test cases or vectors they ran when modifying their Verilog or VHDL code.

In summary, the 1990's were a decade of increased practicality where researchers demonstrated that their regression test selection approaches were applicable to 'real' code. For the big industrial research labs, it was a time of building the tools that could automate the most promising of these research techniques and exploring the transfer of this technology to its business units to provide a competitive advantage.

4. 2000s: Large Scale Application

By the start of the new millennium, the focus of most industrial research labs was shifting to more development work and budgets were being curtailed. Developing leading edge testing technologies, such as those associated with regression test selection, were hard to fund. Only one major player with sufficient resources remained to continue the efforts – Microsoft.

The company had developed an in-house toolset known as Magellan with the help of a research group within Microsoft Research [23]. This industrial-strength toolset included a tool known as Echelon (now called Scout), which codified a greedy algorithm for selecting tests, vindicating the minimization approach. The effort was being driven by the need that the number of automated test cases and thus regression testing costs were exploding. Running and analyzing the results from frequent test passes, whether automated or manual, was very time-consuming and costly.

The Windows Sustained Engineering (SE) team was one of the first teams to apply this in-house regression test selection toolset. The team was responsible for validating maintenance changes to large operating system code bases after they had shipped. Today, these include Vista and Windows 7, together with corresponding Windows server operating system releases. Test suites for these operating system releases run into the hundreds of thousands of tests, both automated and manual. Correspondingly, test passes can take weeks to complete.

By applying the Magellan Scout tool, this team was able to significantly reduce the number of regression tests being rerun, making their maintenance update cycles much faster and more efficient. The team also found that selecting minimal sets of tests did not necessarily affect safety in terms of detecting fewer faults than running all tests again.

Over the years, other teams within Microsoft have also explored the technology with success [24]. The Scout tool keeps evolving to the point where many of the limitations that were discussed at the start of this paper, are no longer valid. A good example of this is the way in which tests are selected. In the latest version of the Scout tool, the original heuristic has been replaced by the latest generation of Microsoft model solvers, which are now capable of solving large and complex linear programming problems in reasonable timeframes and with optimal results. This enables ever larger Microsoft test suites to be processed.

5. 2010 and Beyond

Today, regression test selection lives on and its prospects look promising. With the growth of online services and the need for moving software quality upstream in the development process, regression testing is being explored in the context of *pre-checkin validation*. Developers are creating large numbers of unit test cases, which they cannot all run, before checking in their code. Check-ins are frequent and need to be fast to enable developers to quickly implement and test their features. Regression test selection tools, such as Scout, could provide developers with a viable option of validating only their changed code and reducing the number of unit tests they need to rerun.

The concept of regression test selection has also found its way into Microsoft products. For example, Visual Studio 2010 and 2012 contain a feature known as the Test Impact

Analyzer [25]. Unlike the Scout tool, it does not select a minimal set of tests to rerun, but instead selects an aggregate set, which includes all tests traversing the modified code.

6. Conclusion

This paper has provided a very brief insight into a thirty year history of regression testing, highlighting major research directions, milestones, tools and researchers. It chronicles the evolution of regression testing from research to large-scale industrial application.

7. Acknowledgement

I would like to thank Curtis Anderson, Principal Test Manager of Office Engineering Test, Tara Roth, Corporate VP of Office Shared Services for their ongoing support. My sincere gratitude also goes to Kurt Fischer, Executive Vice President EM&I, my Ph.D. supervisor David Robson and many of my fellow software testing researchers, colleagues and friends who inspired me to explore the topic, guided and challenged me. Special thanks to Prof. Lee White, Prof. Mary-Jean Harrold, Dr. Tom Ostrand and Dr. Elaine Weyuker.

8. References

1. W. Hetzel, Ed. "*Program Test Methods*", Prentice Hall, 1973.
2. G. Myers, "*The Art of Software Testing*", John Wiley, 1979.
3. H. Leung and L. White,"*A Cost Model to Compare Regression Test Strategies*", Intl. Conference on Software Maintence, 1991.
4. A.P. Mathur,"*Foundations of Software Testing*", Addison-Wesley, 2008.
5. K. Fischer, F. Raji and A. Chrusckicki, "*A methodology for retesting modified software*", National Telecommunications Conference, 1981.
6. K. Fischer, F. Raji and D. Onaszko, "*Software Retest Techniques*", Computer Science Corp., Defense Technical Information Center, 1982.
7. Wikipedia (http://en.wikipedia.org/wiki/Set_cover_problem)
8. P. Frankl and E. Weyuker, "*An Applicable Family of Data Flow Testing Criteria*", IEE Transactions on Software Engineering, 1988.
9. T. Ostrand and E. Weyuker,"*Using data flow analysis for regression testing*", PNSQC, 1988.
10. M.J. Harrold and M.L. Soffa, "*An Incremental Approach to Unit Testing during Maintenance*", Intl. Conference on Software Maintence, 1988.
11. M.J. Harrold and M.L. Soffa, "*Interprocedural Data Flow Testing*", Third Symposium on Software Testing, Analysis, and Verification (TAV3), 1989.

12. M.J. Harrold and M.L. Soffa, "*Computation of Interprocedural Definition and Use Dependencies*", IEEE International Conference on Computer Languages (ICCL'90), 1990.
13. W. Landi and B. Ryder, "*A Safe Approximate Algorithm for Interprocedural Aliasing*", ACM SIGPLAN conference on Programming language design and implementation, 1992.
14. Y.F. Chen, D.S. Rosenblum and K.P. Vo,"*TestTube: A System for Selective Regression Testing*", Intl. Conference on Software Engineering, 1994.
15. J.R. Horgan,"*A Dataflow Coverage Testing Tool for C*", Symposium on Assessment of Quality Software Development Tools, 1992.
16. M. Platoff, M. Wagner and J. Camaratta, "*An Integrated Program Representation and Toolkit for the Maintenance of C Programs*", Intl. Conference on Software Maintenance, 1991.
17. M.J. Harrold and P. Kolte, "*", PNSQC,* 1992
18. G. Rothermel and M.J. Harrold, "*A Framework for Evaluating Regression Test Selection Techniques*", International Conference on Software Engineering, 1994
19. M.J. Harrold, J. McGregor and K. Fitzpatrick, "*Incremental Testing of Object-Oriented Class Inheritance Structures*", International Conference on Software Engineering 1992.
20. M.J. Harrold, R. Gupta and M.L. Soffa,"*A Methodology for Controlling the Size of a Test Suite*, Intl.*Conference on Software Maintenance* 1990.
21. J. Hartmann and D. Robson,"*Techniques for Selective Revalidation*", IEEE Software 1990.
22. VN-Optimize (http://www.sereda-tech.co.uk/transeda-vn-optimize.php)
23. A. Srivastava and J. Thiagarajan,"*Effectively prioritizing Tests in a Development Environment*", Intl. Symposium on Software Testing, Analysis and Verification, 2002.
24. J. Hartmann, "*Applying Selective Revalidation Techniques at Microsoft*", PNSQC, 2007.
25. Visual Studio Test Impact Analyzer (http://blog.nwcadence.com/test-impact-analysis-in-visual-studio-11)

Catch Your Own Bugs: Including all Engineers in the Automation Cycle

Laura Bright
Laura_bright@mcafee.com

Abstract

End-to-end automation provides many benefits to developers and QA engineers including faster defect detection and greater product stability. Automation reduces testing overhead and continually monitors overall product quality, enabling developers to quickly identify and fix defects in both new features and regressions.

A challenge to the success of any automation framework is involving the engineers who benefit from this effort. Our experience from past projects has been that most engineers do not monitor automation runs closely due to time constraints or limited accessibility of results. Automation team members are left with the task of interpreting results and filing defects, which places a greater load on the automation team and increases the overall time required to find and fix defects.

This paper presents solutions we have introduced to close this gap in the automation cycle and involve all engineers in the automation process. We draw on experience building end-to-end automation frameworks for two projects and present specific examples of how our solutions have helped developers catch and fix their defects in a timely manner. Our solutions include the following:

- Automatic automation triggers - every time a new build is ready, the automated tests begin executing immediately.
- Email notification of results - An automatic email summary of test results is sent to all team members after every run, ensuring visibility and a prompt response.
- User-friendly UI for result reporting and analysis - Engineers do not need detailed knowledge of the underlying framework implementation to view results and identify the root cause of a defect. Once a defect is fixed and a new build is available, the automation suite will execute automatically and verify the fix.
- All engineers actively contribute to the framework through test script authoring, feedback, and maintenance.

To date, our automation has helped developers and QA catch 19% of the total defects in our current project. Our daily automation runs and reporting solutions have greatly increased developer and QA involvement in automation efforts and improved the stability of our weekly builds.

Biography

Laura Bright is a Software QA Engineer at McAfee where she has led automation framework development efforts for several endpoint security products. She has an A.B. degree from Dartmouth College and a Ph.D. degree in Computer Science from the University of Maryland College Park.

1 Introduction

An end-to-end automation framework provides real-time product quality monitoring and ensures that new issues and regressions are detected in a timely manner. Automatically executing a suite of functional tests on every new build provides immediate feedback on the effects of changes to the code and determines if a build is suitable for further testing.

A challenge to the success of such a framework is enabling all stakeholders to promptly interpret test results and respond to issues. Handling new issues requires sufficient information to determine if a failure is caused by a product defect, and if so, file the defect so it can be fixed as soon as possible. A summary of which tests passed or failed provides a high-level overview of the health of the product, but does not provide sufficient information to identify the cause of failures. If the automation results are not easily accessible and understandable by all team members, team members must rely on automation engineers to interpret test failures and file defects and the cycle is not fully automated. Further, all team members cannot be expected to continually monitor automation results, and should be notified automatically when new results are available.

This paper presents efforts by the McAfee Endpoint Security Automation team to close this gap in the automation cycle and enable developers to catch their own defects.

Our specific goals include the following:

- **Keep all team members in the loop** – All developers, QA engineers, and managers involved with the project should always be aware of the latest automation results.
- **Minimize overhead for automation team** – All stakeholders should be able to drill down and analyze test results without requiring help from an automation engineer.
- **Ease of writing, maintaining, and understanding test scripts** – All developers and QA engineers should be familiar with the automation scripting language and process. This not only allows everyone on the team to contribute to automation efforts, but also improves their understanding of how scripts work so they can easily troubleshoot issues.

We present our experiences extending the automation framework and processes to increase the involvement of all developers and QA engineers in monitoring and interpreting automation results. We implemented new features in our existing automation framework to improve the effectiveness of the automation result reporting and provide detailed information about every test executed. We also discuss increasing involvement of both developers and black box engineers in the script writing process and show how a tool that is easy to learn has many uses beyond the existing automation framework.

Section 2 presents an overview of the team, process, and product, as well the limitations of previous automation frameworks. In Section 3 we provide an overview of the automation framework and present the features that were added to help developers find and fix their own defects. Section 4 presents results including examples of increased developer involvement and defects that were fixed early, and Section 5 discusses future directions and concludes.

2 Background

This section presents an overview of the product, team, and test cases for the automation project presented in this paper.

2.1 Products and Team Structure

The McAfee Endpoint Security Automation team handles test automation for several anti-virus and firewall products. Previously the team built an automation framework for the MOVE (McAfee Optimized for Virtual Environments) product. The current automation effort is testing the latest release of the enterprise endpoint security product. The product has several anti-virus and firewall features including:

- On-access scanner
- On-demand scanner
- Outlook and Lotus Notes Scanners

The development and QA teams are evenly distributed between offices in Beaverton and Bangalore, with automation engineers on the QA teams in both locations. An automation rig is maintained at each location and all required platforms and configurations are tested at one of the two sites. Due to the geographic distribution of the teams, code changes occur 24 hours a day and it is important to catch defects as soon as possible. If an engineer at one site introduces a defect and does not detect it before the end of the day, team members at the other site may need to wait a full day before the engineer fixes the defect if they are unable to fix it themselves and the productivity of the entire team is impacted.

2.2 Build Process

A central build server automatically generates one new build per day. In addition, any team member can request a new build at any time, for example to test new changes that have been checked in. Every build has its status recorded in a central database to identify its status. At least one build per week is marked as "Released to QA" (RTQA) and is used by the entire QA team to execute their test suites. It is important to have a stable RTQA build, otherwise QA team members may not be able to execute their tests in a timely manner.

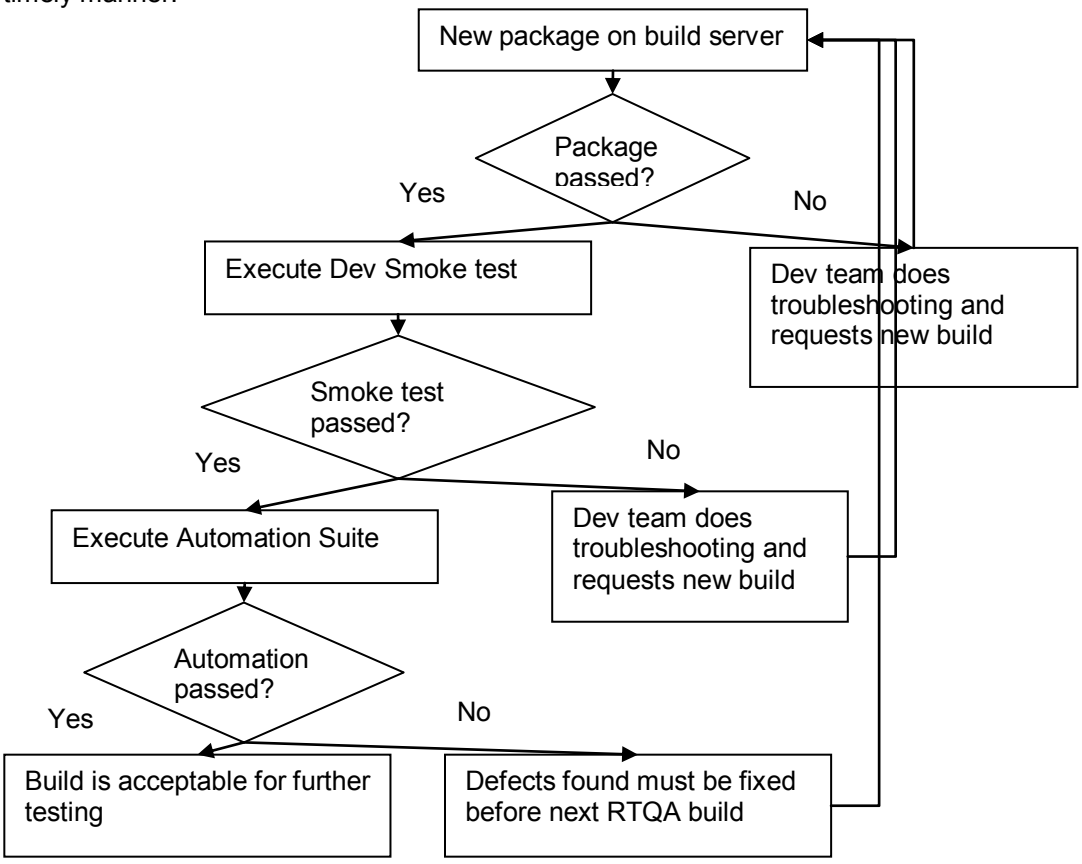


Figure 1: Build Process Flowchart

The build process is outlined in Figure 1. An automatic smoke test created by the development team launches on every successful package to verify basic product functionality. If the smoke test passes, the status in the build server database is updated to "Dev Smoke Test Passed" and the build is ready for additional automated testing. The end-to-end automation rig will automatically begin executing tests on every build that passes the smoke test. If the automation run passes, the build is acceptable for further

testing by the entire QA team. If the automation run fails, any defects that are caught are expected to be fixed no later than in the next RTQA build, sooner if the defect blocks other QA tasks.

2.3 Test Suites

The Endpoint project includes over 4300 test cases, more than 1200 of which are automatable. All tests were audited by members of the automation team to identify whether or not a test case is automatable and the relative automation priority of tests. Automated tests are divided into Build Verification Test (BVT) and Functional Verification Test (FVT). BVT tests cover basic functionality and execute automatically on every nightly build. FVT tests cover product functionality in greater depth and execute on every RTQA build.

2.4 Challenges

Several challenges drive our efforts to improve our automation framework and include all stakeholders in the automation cycle:

- **Frequent code changes and builds** – During an iteration there is a high degree of code churn and significant changes to the code may occur on a daily basis. There is at least one build per day and often several builds if developers are checking in and testing their changes. Manual black box testers cannot keep up with this build frequency, however it is important to catch new and regression defects early so they can be resolved before the next RTQA build.

- **Dependencies on other products** – The Endpoint product consumes and interacts with several other McAfee products, for example the anti-virus engine. Each product has its own QA team and automation setup to monitor product quality, and new milestone builds of dependent products are automatically added to Endpoint product builds when they become available. However, even if a dependent product passes its own tests, changes to the product may impact features of the Endpoint product, or features of the Endpoint product may break due to a previously unknown defect in the dependent product. In addition to detecting defects due to changes in our own product, it is important to detect defects in dependent products so they can be addressed in a timely manner.

- **Geographically distributed team** – With two sites on opposite sides of the globe, product development and testing efforts are continuous. The start of the work day in Beaverton coincides with the end of the work day in Bangalore, and vice versa. This arrangement ensures nearly 24-hour coverage, but has the disadvantage that team members at one site need to wait a day if they need assistance from someone at the other site. For example, if a developer at the Beaverton site checks in code that breaks a product feature, the team in Bangalore may identify a defect but will need to wait until the next day until it is fixed if they are unable to fix it themselves. To maximize productivity, it is important to find and fix defects as soon as possible so team members at the other site will not be impacted.

2.5 Limitations of Earlier Automation Efforts

Our past experience on previous product releases has shown that automation results are not closely monitored by most team members outside of automation. One reason for limited stakeholder involvement is lack of time. Automation runs execute at least once per day, and most stakeholders do not have the time to review results on the web server regularly. Also, most will not think of checking the server unless they receive regular reminders or are monitoring a specific issue.

A second reason is limited access to troubleshooting information and difficulty of interpreting the information. In our previous projects, a web server with the number of tests that passed and failed on every run was available to all stakeholders. While this web server provides a good overview of product

health, it does not provide sufficient information to troubleshoot the root cause of failures. Selected log files from every automation run were made available on a file share, but interpreting these files often required detailed knowledge of the automation framework.

As a result of these limitations, developers and black box engineers typically relied on an automation team member for the status of the latest run rather than monitoring results themselves. Automation engineers had to sift through the logs to identify the root cause of issues and file defects. This process placed an undue burden on automation engineers and increased the length of time required to troubleshoot defects. It also did not fully maximize the benefits of end-to-end automation since intervention from automation engineers was needed. Thus, a major goal of our current automation efforts was to extend the framework to provide detailed information from all automation runs, and provide regular summaries to all team members about the status of current automation runs.

3 Automation Framework Overview

This section presents the architecture of the automation framework and describes specific features we implemented to make the framework accessible to all team members.

3.1 End-to-end Automation

Figure 2 shows the architecture of the end-to-end automation framework. A script that monitors the build server executes continuously on the automation controller. This script queries the build server database every few minutes. Every time the script detects a new build in the database with a status of “Dev smoke

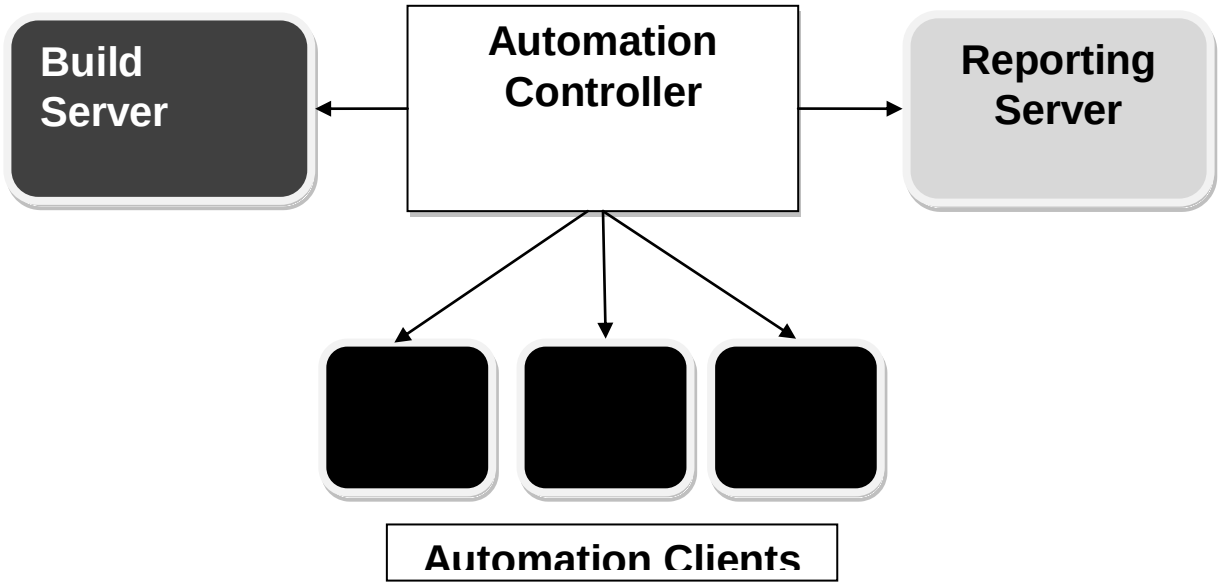


Figure 2:Automation framework architecture

test passed" it launches the automation run. The automation executes on a VMWare ESX server with several client images. Every time a new build is available, the automation controller reverts all clients to a clean snapshot, installs the new build on all clients, and launches an automation batch script on each client to execute all BVT test cases. As tests complete, their status is updated on a web server to indicate PASS or FAIL. When the batch run completes on each client, all log files are copied to the reporting server and user-friendly automation reports are generated on the reporting server. An automatic summary email is sent to all stakeholders with a list of failed tests and their components.

3.2 Test Scripts and Functions

The automation suite executed on each client consists of a batch script that runs a set of test scripts. The test scripting framework consists of a library of PERL functions that are called by individual test scripts. Each tests script is a text file with a sequence of function calls, and is parsed and executed by the framework. No detailed programming knowledge is required and any engineer can read and write test scripts. Our goal was to make the framework as easy to learn as possible and goes along with our efforts to incorporate all engineers in all stages of the automation cycle.

```
SetLogLocation ‘1234.txt’
CreateEicars ‘eicar.exe’
VerifyLogTextC ‘Deleted.*eicar.exe’ 1 ‘1234.txt’ 1
VerifyFileExists ‘eicar.exe’ 0
```

```
1 PASS Log location set to 1234.txt
2 PASS Eicar file eicar.exe successfully created
3 FAIL       Specified text ‘Delete.*eicar.exe’ not found
4 PASS File ‘eicar.exe’ does not exist. Expected.
```

Figure 3:Test script and log example

Each framework function call is a test step and reports PASS or FAIL to a log file along with the details of the test step. . If one or more steps fail then the test case fails. Users can review the automation logs to see which test steps failed and identify the cause of the failure. An example is shown in Figure 3. At the top is a simple test script that changes the product log name, creates a sample file to be detected by the on-access scanner, and verifies that the file was deleted by the product and the detection was logged. On line 3 of the sample automation log, the specified text is not found in the product log, so there may be a logging code defect.

In addition to improved understanding of test results, a benefit of an easy-to-use test script language is that all developers and QA engineers can contribute by writing test scripts as well. We discuss these efforts further in Section 4.2.

3.3 Email Notifications

Automatic email notifications are sent after every run and include a list of tests that failed on each platform as well as a summary of the number of test failures per component. The email also indicates clearly if the install failed on a particular operating system. Since the dev smoke test only covers a single platform and installation failures often apply to a particular class of operating system, the nightly runs are helpful to detect installation issues that may have been missed by the smoke test. An example email summary is shown in Figure 4.

Since the automation runs continuously, a run may complete during nighttime hours at one of the two sites and many automation engineers may be out of the office. Automatic emails ensure that stakeholders are immediately aware of automation results even if no automation engineers are present. In some cases,

developers can be notified of new issues and identify the root cause before any automation engineers have had a chance to look at the results. We present examples in Section 4.

All managers and team members at both sites receive these emails. Initially some team members did not closely monitor the emails, but over time nearly all team members have started to monitor them and respond promptly to unexpected failures.

```
Results for 2K3ER2:
        0 tests passed and 1 failed
       Failed tests:  19045  ***** NOTE: Install failed on 2K3ER2! *****
Results for 2K8R2:
        115 tests passed and 7 failed
       Failed tests:  11336 8415 8416 11016 7544 7531 9580
Results for Win7x86:
        129 tests passed and 8 failed
       Failed tests:  11336 8415 8416 7544 7531 7712 11373 9580
Results for XP3:
        125 tests passed and 12 failed
       Failed tests:  11336 8415 8416 7544 11016 10726 9304 7712 9640 9560 9793 9580

Failed tests by component:
AV-Outlook Email Scan=>OnDelivery ScanSettings: 1
AV-OnAccessScan=>Exclusions: 5
AV-OnAccessScan=>Process Classifications: 2
AV-OnDemandScan=>ScanSettings: 9
AV-Outlook Email Scan=>OnDemand ScanSettings: 5
VSE/Functional/AV/OnAccess Scan/Process Classifications: 1
AV-LotusNotes Email Scan=>OnAccess ScanSettings: 2
Common-Installer=>Default Install: 1
AV-ScriptScan=>Exclusions: 1
AV-Quarantine Manager=>Populate: 1
```

Figure 4: Example email summary

3.4 Reporting Web Server

The reporting web server contains detailed automation logs that indicate which steps of each test script passed and failed, as well as a graphical interface to summarize this information and allow users to drill down to individual results. Historical data is saved from earlier builds and iterations, so users can compare earlier runs against the current run. An example web report is shown in Figure 5. The drop-down menus at the top of the page allow users to select the build and iteration (the default is the most recent build). The automation summary shows the percentage of tests that failed on at least one platform, and the percentage of tests that passed on all platforms. If you click on an individual test, it displays the test results on each platform, and you can click on an individual platform to see which test steps failed as we will show in Section 4.

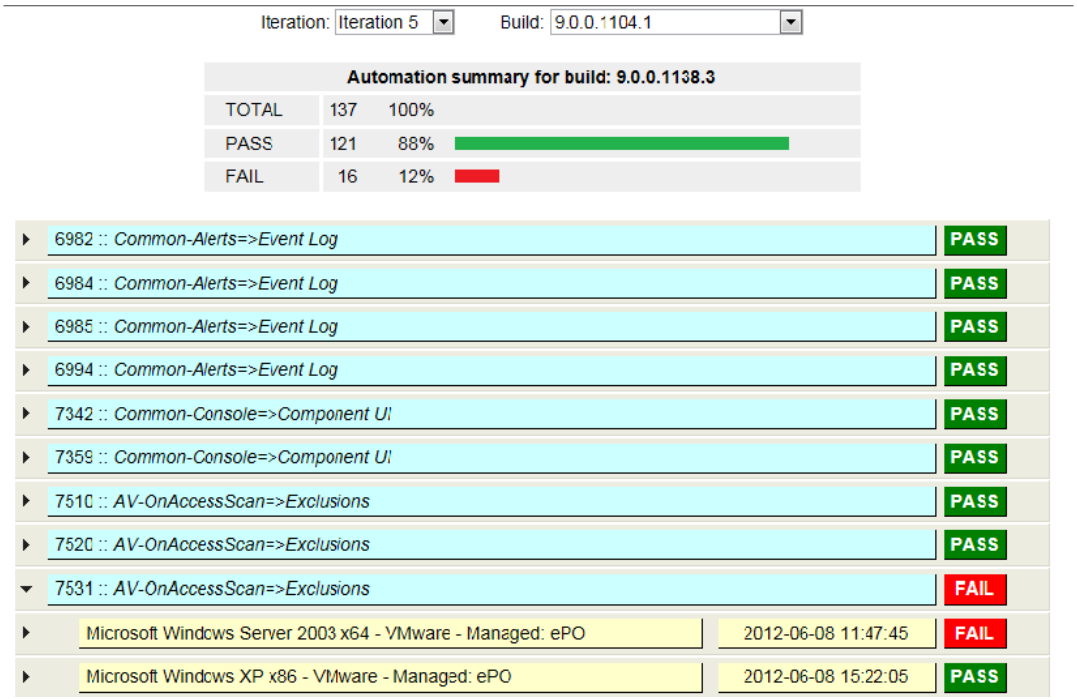


Figure 5: Automation report web server

The server also stores the following files from every automation run to help with troubleshooting. Access to these files is linked from the web reports so users can easily drill down for more information.

- **Product logs** – All logs from the product. This includes install logs which are helpful if the install fails. Logs are renamed in each test case to include the test id in the name so they can be easily identified.
- **Product configuration** – Each test saves a text file which contains the product settings at the time the test was run. If a test fails because a product feature did not perform as expected, these files are useful to verify that the product settings were as expected and determine if the failure is a product defect or test script error.
- **Event logs** – Windows event logs are saved and can be used to verify that product-related events are reported properly, or for troubleshooting if any unexpected system behavior occurs (e.g., a process crash).
- **Crash dumps** – If any process crashes during test execution, a WinDbg crash dump is automatically generated and copied to the report server at the end of the test run

3.5 Process

As a team we adopted the following practices for responding to test failures. Nightly BVT runs execute only on stable test scripts that are expected to pass, so automation failures are rare and failures normally indicate new or regression defects that should be filed and fixed in a timely manner. In some cases a developer can identify the root cause of the defect using the web server reports, and in these cases they are expected to fix the issue immediately even if a defect has not yet been filed. If assistance from an automation engineer is required, the engineer will analyze the failure and file a defect. If it is determined that automation framework logging could be improved to make troubleshooting easier, a defect will be filed against the automation framework and an automation engineer will make the fix. We present specific examples of defects that were found and fixed by developers in the next section.

4 Results

This section outlines success stories of our framework including specific examples of defects that were found and fixed by our framework. Some major benefits of the framework include:

- Earlier detection and fixing of issues – Automation runs occur 24 hours a day whenever a new build is ready, so results can be immediately reported to stakeholders. Several defects detected by nightly automation were fixed by developers before QA had time to file a defect. Many additional defects (>50) were observed immediately and prompted developers to ask the automation team about the status of the build. For example, there were 21 installation defects that indicated prompt action was needed. The automation runs also provided a good status monitor and provided automatic verification of bugs that were fixed.
- Information from all runs is stored on a server accessible to entire team. Historical information is available to monitor product quality over time.
- Reduced time in testing cycle and increased testing frequency
- Improved product stability – defects detected in nightly runs are fixed before the next RTQA, leading to better quality of RTQA builds and lower probability that an RTQA build will be rejected.
- Increased defect detection – There have been 19% of total defects found early through automation. Many of these defects were crashes that were difficult to reproduce in a manual environment, and 51% of Severity 1 defects were found through automation.

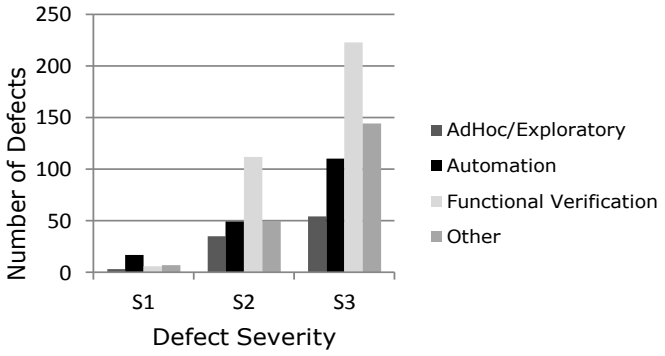


Figure 6: Defect severity by methodology

	S1	S2	S3	S4	Total
Percent defects found	51%	20%	20%	10%	19%

4.1 Success stories

This section presents a few examples of new defects and regressions that were rapidly detected by the nightly BVT automation and reporting framework. The nightly automation suite contains only tests that pass reliably, so any failures are considered regressions that need to be addressed promptly.

4.1.1 On-Demand Scan Regression

A developer in Beaverton made some changes to the configuration of which folders to scan in on-demand scan tasks. After these changes were checked in, three on-demand scan tests that had been passing previously started failing on the next build. The developer who had made the change observed that the failed tests were related to the on-demand scanner so they were likely related to the changes.
The web report for one of the three failed tests is shown in Figure 7. This test script configures an on-demand scan on a mapped network drive, and verifies that the samples on the drive are deleted and the detections are logged. In this example, the log indicates that the sample files were not logged or deleted, indicating that the on-demand scan did not scan the proper location. The developer promptly corrected the issue.

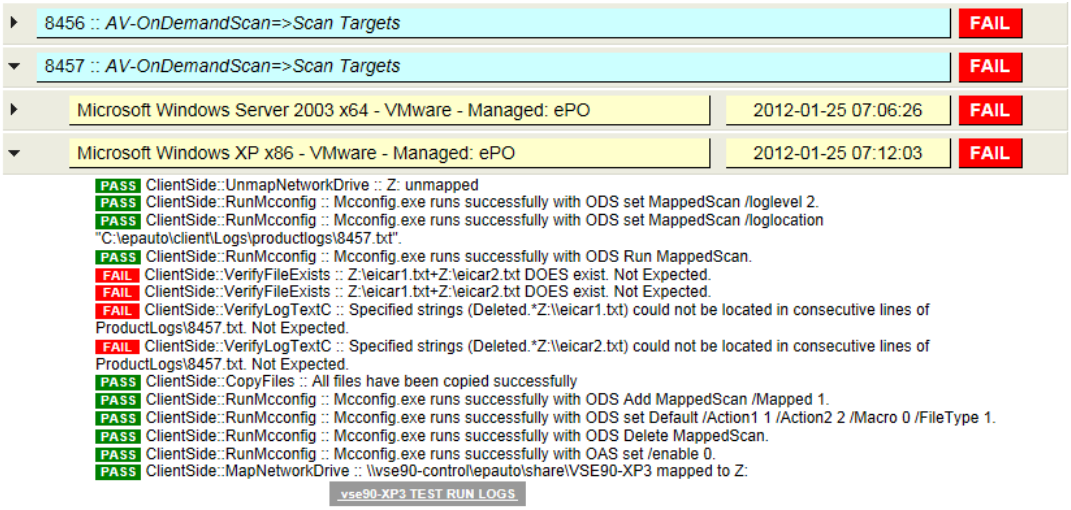


Figure 7: On-demand scan report indicating failure to scan mapped drive

4.1.2 Lotus Notes Scanner Regression

A developer in Bangalore checked in some changes to the Lotus Notes scanner code and requested a new build on the build server. The automation rig in Beaverton automatically launched the BVT suite on the new build, even though it was nighttime in Beaverton and no engineers were present. After the run completed, an automated email summary was sent to the entire team indicating that all of the Lotus Notes scanner tests had failed.

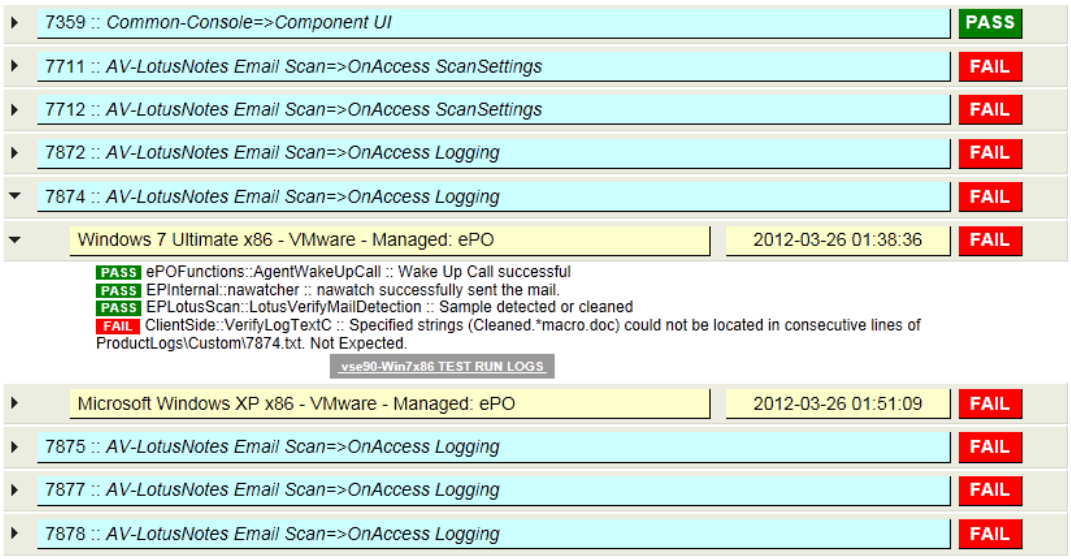


Figure 8: Web server report indicating Lotus Notes Scanner logging failures.

The developer who had checked in the changes immediately determined from the email message that the changes had broken some of the Lotus Notes Scanner functionality. Further investigation of the web server results indicated that the logging functionality was not working as expected (Figure 8). This log shows that the LotusVerifyMailDetection function passed, indicating that the sample was detected by the scanner, but the VerifyLogTextC failure indicates that the detection was not properly logged. The developer promptly notified the team that he was aware of the issue and was working to fix it. When automation engineers arrived at work in Beaverton the next morning, the defect had already been fixed and all the Lotus Notes scanner tests passed on the next automation run.

4.2 Test Scripting Efforts

Including developers and manual testers in the test scripting process has many benefits including increasing the number of automated tests and improved understanding of automation results.
If everyone on the team has a good understanding of test script structure and framework functions, it is easier to interpret automation results without relying on assistance from automation engineers. Also, manual testers can leverage the framework functions to write scripts that assist them with repetitive testing tasks. Scripts can also be a useful way to reliably reproduce defects. We briefly discuss two efforts to include developers and manual testers in the test scripting process. We plan to continue these efforts in the future and train all developers and QA engineer to write test scripts.

At the Bangalore site, the automation engineers worked with developers to automate a set of tests. Each developer was assigned around 5-10 test cases to automate and had assistance from automation engineers to answer questions, fix bugs, and add new functionality to the test framework as needed. The developers have strong programming skills and learned the framework quickly. However, there was a learning curve to understand test script best practices such as setup, cleanup, and appropriate conditions to verify. All of the scripts were reviewed by automation engineers and after minor fixes most were added to the FVT automation suite.

The Beaverton automation team has also successfully trained two QA team members to use the automation framework. These efforts started when QA engineers completed their iteration tasks early and had time to devote to other tasks. Since there are hundreds of automatable test cases, the automation team asked them to help write test scripts for some of these tests. Beaverton automation engineers assisted QA team members with setting up an automated testing environment and gave them a brief tutorial and script examples. After writing a few test scripts, QA engineers were quickly brought up to speed on the automation process and best practices. The engineers have automated a large number of

tests and continue to actively contribute to automation efforts. Training QA engineers to automate tests has significantly increased the productivity of the automation team and is a good way to leverage the spare cycles of the engineers when they do not have higher priority QA tasks.

5 Conclusions and Future Directions

We have shown the success of preliminary efforts to include all team members in automation and have shown ways that all engineers can contribute to and benefit from automation. The automation and reporting framework has enabled rapid defect detection and 20% of product defects have been detected by automated tests.

Our future plans include improved reporting features including linking failures to known defects and identifying new and unexpected failures. This information will help all team members better understand test results and prioritize troubleshooting efforts. We would also like to link test to specific product features to help team members more quickly identify the root cause of failures. The goal is to identify the components and functionality covered by each test script and store this information in a database. When one or more tests fail unexpectedly, we could query the database to determine which components may have caused the failure. Similarly, if a developer checks in changes to a component, they could determine which tests are affected by the change and monitor the results of those tests in the next automation run. Finally, we plan to continue training developers and manual testers to use the automation framework and hope to continue to increase the productivity of the entire team through automation.

Acknowledgments

Praveen Soraganvi provided valuable feedback on the topic and scope of this paper and on an earlier draft. Everyone on the Endpoint Security Automation team including Keith Albin, Steve Nguyen, Dale Wacker, Jenny Yu, Anand Iyer, Sudhindra Kembhavi, and Amit Patel contributed to the design and implementation of the end-to-end automation and reporting framework described in this paper. We thank everyone on the VSE development and QA teams and the MOVE development and QA teams for their feedback on the automation framework and for successfully incorporating the framework into their projects.

Are We Failing at User Interface Test Automation?

Jim Holmes
Jim.Holmes@Telerik.com

Abstract

Time and time again over the last several years I've run in to individuals, teams, or organizations that are struggling or outright failing with test automation at the User Interface level. Some of these people are members of teams that are writing great systems. If they're able to write great systems, where's the disconnect with having so much difficulty around test automation? These issues aren't isolated ones; rather, they're much more common than many in our industry are willing to admit.

This paper and presentation focus on some of the main reasons for difficulties in good test automation: lack of understanding of fundamentals with the technologies, failure to recognize that automation efforts are software engineering efforts, and a failure to focus on high-value automation over low-value tasks. This paper is completely technology agnostic—these issues cut across all domains.

In this paper I will discuss ways to avoid these pitfalls, and ways to ensure your teams and organizations are able to identify and overcome these challenges. We'll walk through the importance of approaching automation as a whole-team effort. You'll learn to identify candidates for good automation and which task to avoid. You'll also come away knowing why it's critical to have a solid understanding of the technologies involved.

Finally, you'll take away why it's so critical to keep your team and organizations constantly assessing their efforts, and adjusting where needed – ***JUST LIKE GOOD SOFTWARE TEAMS DO.***

Biography

Jim Holmes is the Test Studio Evangelist at Telerik. He has over 25 years in the IT field in positions including PC technician, WAN manager, customer relations manager, developer, and yes, tester. Jim has held jobs in the US Air Force, DOD sector, the software consulting domain, and commercial software product sectors. He's been a long-time advocate of test automation and has delivered software on a wide range of platforms. He co-authored the book Windows Developer Power Tools and blogs frequently at http://FrazzledDad.com. Jim is also the President of the Board of Directors for the CodeMash conference held in the middle of winter at an indoor waterpark in Sandusky, Ohio.

1 Are We Failing at UI Automation?

I define success in User Interface (UI) automation as building a suite of sustainable tests which are part of a team's overall strategy of delivering great value to our customers. Given this yardstick, "Are we failing?" has several answers depending on how you look at the question.

No: Great tools around UI automation are seeing exciting adoption rate increases. Selenium/WebDriver, Watir, and commercial tools continue to see growth across the industry. Selenium has eclipsed Quick Test Pro in jobs listings for the first time ever. More organizations and teams are beginning to understand the importance of a whole-team approach to test automation.

Maybe: In the software development domain we've seen an explosion of extraordinary methodologies such as Test Driven Development (TDD) or Behavioral Driven Development (BDD). We've seen a tremendous amount of books, articles, conference presentations, and training courses around these methodologies. The software craftsmanship movement is fostering lots of discussion around how to best build high-quality, valuable software. All that's great, but where's the corresponding content and discussions around UI automation?

Yes: Particularly in regards to enterprises and with newcomers to automation. Enterprises are often struggling with difficult development processes and environments. Automation can be difficult to implement in these places because of a lack of skills and poorly set expectations with the teams, stakeholders, and other management. Newcomers to UI automation are often lacking fundamental skills and knowledge, and are often put in positions with no useful mentoring to help them succeed.

These failures, and their impacts, are traceable to a number of critical human and cultural factors. There are also a number of problems relating directly to technology.

We can fix these problems, but we need to first acknowledge it's going to be a long haul and will require a lot of effort. We can look to the software craftsmanship movement as a great guide to help us bring the automation segment of our organizations up to a level of great proficiency, which will enable us to deliver even better software to our stakeholders and customers.

2 Impacts of Failure

Roy Osherove's *Art of Unit Testing* (Osherove 2009) opens with a preface about a project Roy was leading some years ago. Roy's team was new to TDD, and had decided to go 100% TDD with a new customer's project. Unfortunately, the team didn't have the skills to create good automated unit tests, and they quickly lost control of their test suite. Soon they were spending more time fixing their tests then actually building and delivering value to the customer. As a result the customer pulled the plug on the entire project. Please re-read that: the customer pulled the plug on the entire project. Not just the testing portion, ***the entire project.***

Automated tests, especially automated User Interface tests, can quickly spiral out of control into a huge morass of pain and frustration. It doesn't take much for a team to create a brittle, cumbersome, overly complex suite of tests that requires huge amounts of time for maintenance, leaving far too little time for actually delivering value to the customer.

Not only does this failure suck the morale out of the team, it can also lead to a fatal breakdown of trust with your stakeholders. The customers don't see value being delivered on a regular basis, and they start to question the value of any automation at all. It's easy to understand this point of view: "You're spending more time fixing your tests. Why not skip those and just write more code for the system you're supposed to be delivering to me?"

Worse yet, that breakdown of trust is rarely limited to the automated testing suite. The customer/stakeholder now begins to fear your team doesn't have the right skills or discipline to help solve their critical business needs. Your project may be in jeopardy because you're not able to get your automation efforts rolling along smoothly.

3 Failure from Human Factors Influence

Test automation is a complex domain, and there are many factors in our industry's failures with it.

As with every complex problem, human factors contribute to many of the largest issues. First, teams (I include stakeholders and management in this too) on projects don't realize or acknowledge that UI automation is at its heart a software engineering effort. UI automation requires skilled people, hardware, and time to get right. It's not something that's easily bolted in to a project after it is nearly completed. This same concept can be extended to the perception of Quality Assurance/Quality Control as a secondary effort in projects.

Additionally, manual testers can often be hesitant to get involved with automation efforts. Their reluctance is nearly always traced back to FUD, a great acronym used for Fear, Uncertainty, and Doubt: they feel automation will put them out of a job, they're fearful about being able to contribute anything of value to automation, and they're scared of having to learn the new skills required for automation.

Another problem involves a long-standing issue with those we look to for help: UI automation has had far too much snake oil sold to the industry from commercial tool vendors, consultants, and thought leaders in our testing domain. Too many commercial vendors tout their toolsets as a panacea and gloss over the long-term investment needed to bring a team up to an effective skill level. Those same vendors also fail to address the long-term view for test suite maintainability as a project ages.

Consultants don't get a pass here, either. Too many consultants have made their living focusing on pushing tools, both commercial and open source. They've pushed these changes in to environments without much thought about the long-term success with the teams that will have to maintain those suites, tools, and frameworks after the consultant has left.

Additionally, a number of thought leaders in the manual and exploratory testing realm speak dismissively of automation because they think it detracts from the positions they've staked out as leaders. This is a rather sad position for these leaders to take because it enforces the misperception that there is a single-facet solution to delivering quality software to our customers. The reality is that quality requires many different aspects from the whole team.

Making everything worse, we in the software realm do a miserable job effectively communicating the time and dollar impacts of UI automation, and we fail in communicating the value of UI test automation to stakeholders. This isn't surprising: we've taken far too long getting our customers and stakeholders to buy off on the value propositions behind quality software design and developer-level testing.

The final nail in the coffin is that, far too often, automation isn't seen as a software engineering problem – **but it is**! UI test automation has often been thought of as a test activity, or a problem solved by tools with no need for true development skills. Sadly, that's far from the truth. We need to acknowledge that test automation is a true software engineering effort and requires the involvement of the whole team from stakeholders through developers to testers.

4 Failure from Technology

The points above are all large factors in the success or failure of UI automation, but they're not the largest problem. At the heart of automation woes is one concise issue: It's a hard problem to solve!

UI automation hits a huge, complex array of facets across the entire project:

- Every piece of technology from the web servers down to the database for web applications
- Unfriendly platforms for automation in desktop applications
- Test-hostile UI designs
- Environmental issues (where do we test, and how often?)
- Hard to learn languages, platforms, frameworks, and toolsets

It's not hard to see that automation can be a struggle given the wide range of challenges I've listed. These are the same sorts of problems we run in to on the development side of projects. Developers have long had some great guideposts to look at in their domain such as Steve McConnell's *Code Complete* (McConnell 2004), Andy Hunt and Dave Thomas's *The Pragmatic Programmer* (Hunt 1999), or any of the great works by Robert "Uncle Bob" Martin. Sadly we don't have any similar works to look to for guidance around UI automation. Several works, like Fewster and Graham's *Software Test Automation* (Fewster 1999), come close, but they're not widely enough read, nor are they impactful enough to be considered in the same light as the aforementioned works.

Unfortunately, too many testers don't understand the underlying technical aspects of the systems they're working on, nor do they understand basics about automation. Time and time again I see teams of developers and tester who are missing the point when it comes to creating stable, repeatable tests. Too often they're relying on brittle, hardwired locators using XPath instead of looking to more appropriate things like ID values. I also see teams littering their test suites with pauses and fixed delays in order to overcome common problems with dynamic content from AJAX or similar service calls.

Testers who have a great grasp of focused, granular manual tests seem to lose their minds when working with automation. Automated tests cross numerous concerns, perform checks across multiple situations, and weave in to muddled, confused scripts hundreds of steps long.

Both the above situations lead to extremely brittle tests which fail intermittently and are incredibly painful to maintain when the system changes. Both situations tie directly back to concepts central to good software design: conciseness, simplicity, and solid design. These same general problems have been prevalent in the development side of our industry for some time; however, there's been a significant amount of attention brought to these problems over the last five to ten years from the viewpoint of software developers.

When I ask thought leaders in the testing industry about these same problems I tend to get blank stares and responses like "That's a problem? We solved that with our team ages ago." This really shows me that long-time practitioners of test automation have completely lost sight of the gap between their skillset and the sad state of the rest of the industry.

5 How to Fix It

First off, there aren't any quick fixes for this large problem. It's actually a number of problems, and most of them trace directly to human factors and cultural issues. Problems with those root causes are never easily fixed. We have to set our expectations that this will be a long-term effort with a lot of setbacks along the way. We must be willing to adjust failed approaches and think up new ones.

We also have to acknowledge that this problem has been around since the start of test automation. In 2001, Dawn Haynes wrote a tremendous article for Rational Edge (Haynes, 2001) highlighting many of the same exact problems we're still facing 11 years later.

A great starting point for this long haul is acknowledging that automated testing is a software engineering effort, not some after-the-fact bolt on QA task. Simply looking at the problem in this light helps us realize we need to look outside the narrow silo of QA/testers/whatever and bring in the rest of the project team to help us succeed.

Using a whole team approach to automated UI testing is critical to successful efforts. Bringing in all voices to the automation work gets us invaluable input from many viewpoints:

- **Stakeholders** identify what their most critical risk areas are. This helps the entire team understand what to focus automation efforts on. “Show me the money” tests might be important for some stakeholders; validating cross-browser functionality may be more important for others.
- **Project Managers/Scrum Masters/etc.** have a broad picture of the workflow and can contribute greatly to ensuring the right priority on value decisions around test automation happen. Additionally, they’re roadblock destroyers who can help when more resources and people are needed.
- **Developers** need to be pairing up with testers to create UI automation. Developers’ skills for software design apply directly to ensuring tests are avoiding duplication and are well designed from a software view. Moreover, developers can quickly flip over to the system under test and alter the UI as needed to improve testability. Additionally, they can assist with writing backing APIs to help with setup, teardown, and configuration steps.
- **Testers** need to be involved on nearly every project larger than a few pages for a static website. Developers and PMs don’t have the domain knowledge or skill to create high-quality tests that aren’t simply “happy path” tests.

Bringing the whole team in to automation efforts helps emphasize that automation is a software engineering task. Time and resources (*resources are **not** people!)* need to be allocated to get the tests done in the first place, and time needs to be set aside for ongoing technical debt payoff and refactoring. Moreover, thinking of automation as a software engineering effort helps us ensure we’re including automation in every step of whatever process the team is using. “Done Done” can’t be “Done” if there’s no automation around the feature!

If we’re buying off on the whole team approach - and we absolutely need to be! - then we need to work hard to bring manual testers in to the fold. As I mentioned earlier, I repeatedly see manual testers who are extremely resistant to automation because of simple FUD. They’re worried they’ll be out of a job, or they’re worried they won’t be able to contribute in a meaningful fashion. The team has to show manual testers their domain knowledge has tremendous value. We have to show them that automation frees them from rote repetition of regression suites and frees them up to spend time doing high-value exploratory testing. They’ll finally be able to put their brains to great use instead of simply following scripts!

Continuing in this educational bent, we must raise the basic skill levels of our folks working with test automation. People working on automating web applications can’t continue without a fundamental understanding of page load lifecycles, DOM structures, element locators, and dealing with dynamic content. Teams shouldn’t be looking to conferences to save them in these areas. Instead, these areas are easily covered through things like brown bag lunches, user groups, meet ups, and many other self-organized opportunities.

This dovetails right in to a larger problem we have across the entire software industry: a miserable state of mentoring. Focusing in on the automation issue, we need the more experienced people on our teams to step up to the plate and mentor those behind them. Empowering teams to get rolling on automation, and supporting them as they work through the difficult learning process is critical. (See the Conclusions section for some specific action items on this topic.)

That said, we need to ensure we’re getting more technical presentations at conferences. Too many well-known conferences are deserts when it comes real-world, take-this-home-and-put-it-to-use practical sessions involving anything around the technical side of testing. Developers need to start submitting talks to conferences traditionally thought of as QA-only domains. People working with test automation regularly need to be submitting talks and white papers about their experiences. Passing this knowledge along to conference attendees will be a great boost to the overall skill level of the industry.

Tool vendors, both commercial and open source, need to step up to the plate, too. Stop pitching the tools and frameworks as panaceas and be forthright about the level of effort that will be necessary. Consultants pitching their efforts around tools, commercial or open source, also need to be much more honest about the long-term impacts on the organization after the consultants leave test suites to the organizations when the projects are done.

Open source leaders need to step back and really focus on getting their projects' documentation up to snuff. Getting workers to write documentation for open source projects isn't ever easy, but it's time to step up to the plate and get usable, current documentation in place that educates newcomers. Simply pointing to Google searches for blogs doesn't cut it any more.

Moving away from the tooling aspect of this and returning to the human factors/cultural issues, I think that we in the testing industry can actually take a lot of heart in the passion, traction, and change fostered by the software craftsmanship movement. This movement has a great many voices and opinions, but at its root is an acknowledgement that things we've been trying haven't worked well, and that we need to get fired up about our work. The software craftsmanship emphasizes team empowerment and responsibility as vectors for getting software built right: care enough about what you do to take time to learn how to do it well. Then do it well.

We can draw off many experiences from the craftsmanship movement, and frankly we shouldn't be trying to generate our own similar movement. Automation is a software engineering effort, so let's just hitch up to the craftsmanship movement and add our voices and experiences in there!

6 Getting Started: A Practical Example

I can look to my own history with automation for guideposts to a successful implementation. (I'll save you the war stories of my various unsuccessful efforts. Yes, those have happened.)

First off, set the tone with the entire team, from stakeholders to your operations support folks. Getting going with automation requires acknowledging up front there will be a learning curve. It also requires acknowledging there will need to be involvement from both testers and developers, regardless of the toolset being used. This is especially true for tools like Selenium IDE or commercial tools seeming to hold the promise of "codeless" solutions for automation efforts.

Once you've got those expectations correctly set, begin with small steps. Have your team identify a few high value test cases to target for automation. Your whole team needs to be involved in this discussion as you select cases and decompose them. Stakeholders should have a say in the highest priority test cases—they've the best exposure in to what the business values. Testers, especially the manual ones, will have great ideas on how to flesh out simplistic cases in to realistic exercises designed to avoid regressions. Finally, developers will know what value they can add by updating the UI where needed, or by creating backing APIs to leverage existing web service or database calls to handle setup and teardown steps.

Work hard to have your developers and testers create tests via paired programming. Both will have great ideas on how to get the best test cases built in the quickest, most maintainable fashion. Having developers assist in this effort can help ensure the cases follow good development practices (granularity, avoiding duplication, etc.). Getting the testers writing tests gets better coverage with more robust cases covering sad paths, odd validations, etc.

Make your automation part of each feature you're working on. "Done Done" shouldn't be "Done" until you have working UI automation cases for it. Don't release new features to production unless you have test automation as part of that work. This isn't an easy thing to do—you'll have to get the entire team to commit to this, and it's extraordinarily easy to let this commitment slip when things get rough.

Above all, you'll need to empower the team with a powerful, passionate mindset for continual improvement of the test suite. We need to look at our test suites just like we do our production codebases: they're best when we constantly pay off technical debt, refactor out poor approaches, and occasionally completely re-architect the solution if needed.

Successful automation projects always look at the test suite just like production code—***Because it is!***

7 Conclusion

Are we failing with UI automation? As I pointed out in the opening, it depends. We have tremendous failures, we have tremendous wins. Successful UI automation is within the reach of every software team. It requires discipline, training, and constant refinement. It also requires the whole team to focus on what the most important aspects of that automation effort are.

If you're involved in even modestly successful automation efforts, share your views. Present at conferences, write articles for magazines, or start a blog. Don't be intimidated by what you think are meager accomplishments—other teams out there are struggling through the same issues you've overcome. They're looking for information on others who've gotten through those struggles. Too often successful teams in our industry forget the number of people still struggling through issues they've solved months or years ago.

If you're looking to start working on UI automation, then get involved by monitoring automation mailing lists like the Selenium user (http://groups.google.com/group/selenium-users) or developer (http://groups.google.com/group/selenium-developers) lists. Grab Jeff Morgan's page-object gem from Github (https://github.com/cheezy/page-object) and start exploring it.

All off this requires us to keep a very patient, long-haul view. It's taken decades for the software development side of our industry to work up to its current state, and I don't think anyone would call it perfect. Instead, we need to acknowledge there's a lot of work to do. Let's be about it!

References

Haynes, D. 2001. "Automated Testing: A Silver Bullet," *Rational Edge*, June, 2001

Hunt, A. and Thomas, D, 1999. *The Pragmatic Programmer: From Journeyman to Master*. Pragmatic Press, 1999.

McConnel, S. 2004. *Code Complete: A Practical Handbook of Software Construction* 2nd edition. Microsoft Press, 2004.

Morgan, J. Page-object Gem, https://github.com/cheezy/page-object.

Osherove, R. 2009. *The Art of Unit Testing: With Examples in .NET.* Manning.

Selenium Developers Google Group, http://groups.google.com/group/selenium-developers.

Selenium Users Google Group, http://groups.google.com/group/selenium-users.

Green Lantern Automation

Sridevi Pinjala
http://www.linkedin.com/in/SrideviPinjala

Abstract

Once upon a time, the average life span for automation scripts that were recorded and played back was until the new user interface replaced the old one or until the HTML ID's were changed during a subtle application face-lift. This kind of change is expected; seldom, once every 5 or 7 years, or frequently with every build.

Major functionality additions, minor bug fixes, elusive changes to improve the response time, etc. all require the testers to spend a lot of time performing regression testing. Build verification and predictable functional tests can rely heavily on automated testing to complete the task. Automated test scripts type in fields, click on buttons, click and wait on links, select from drop down lists, verify text, logos, images present, and compare strings of text or results.

There is a simple way to create a UI/GUI repository in selenium IDE. There is a way all the UI/GUI elements can be named in English (my favorite language). Any changes to the IDs can be tracked and updated and the scripts do not have to be modified or tweaked individually ever again.

Typically, tools that have a repository map such as TestComplete, Quick Test Professional, Rational Functional Tester, etc. collect and store all the GUI elements in the object repository map. When the DOM (document object model) or the HTML ID name or other properties of an element change in an application, this change could be managed / updated via the object repository map.

Automation tools such as Selenium IDE rely on the HTML ID attribute of the User Interface (UI) and GUI (graphical User Interface) to recognize an element. They do not have object repository maps to store all the UI and GUI elements. If the HTML ID changed for any reason, the script death is inevitable. To keep-up with the changes and to avoid script death, each and every script has to be changed.

This paper introduces a concept I call "Green Lantern Automation". I chose the super hero Green Lantern because – Each Green Lantern possess a power ring and power lantern that gives the user great control over the physical world as long as the wielder has sufficient will power and strength to wield it. (Green Lantern) This concept retains the script when minor and major changes occur to the application. We automation test suite developers can control the longevity of our scripts by creating them with appropriate standards. Our standards and conventions are only as good as our ideas. Just like the Green Lantern creates objects with his will power.

Biography

I am a Sr. SQA Engineer at IBM. In 2011 Cynthia Gen (my reviewer) asked me to come up with a similar concept as Green Lantern Automation Framework with open source tools like Selenium and luckily, there is a way to implement the concept with Selenium IDE and also to take it to the next level. More ideas at – http://SrideviPinjala.blogspot.com/

Introduction

This presentation will give step-by-step instructions for creating a Test suite utilizing the Green Lantern concept. A tester can take this concept and utilize it for creating test suites for one's applications.

I will discuss the possibilities and show how a script written for one UI (Gmail) can be used for a different UI (Netflix). I will discuss the recommended naming conventions, standards and methods to be followed while creating a project suite utilizing Selenium IDE. I will also explain the advantages of following these standards and naming conventions.

First step is to specify a location where the test result pictures will be stored. I log messages with pictures rather than go through the log file and find the errors, warning or information messages.

Second step is to identify the elements (text field, link, button, page, text, check box, etc.). The developers usually assign a "logical name" to these elements. I rename the "logical names" with a naming convention.

Third step is to create a data sheet where the data is saved in HTML format. This data will be used by the test scripts as an alternative to hard coded test data.

Fourth step will invoke the URL intended for testing. I like to use one selenium test case solely to invoke the URL that can be replaced and used with other functionality.

Last but not least, the ingredient that adds value to this concept is being able to post a screen shot at every passed or failed step with a meaningful message as its name.

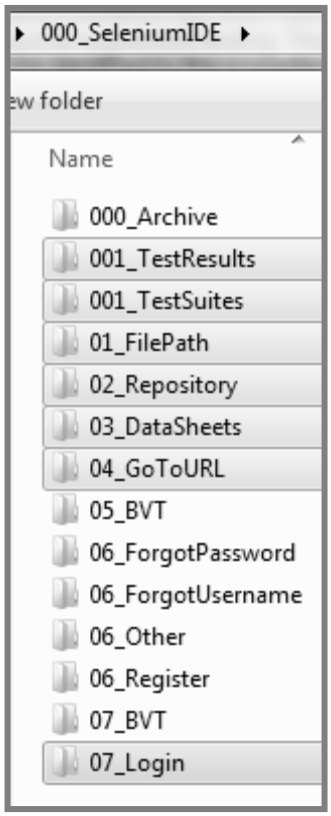
01_filePath, 02_repository, 03_dataSheet (if data is required), 04_GoToURL are the required test cases for this concept. In order to make it easy to remember and add the required test cases, I number the test cases and save them in the same numbered folder. I further numbered the functional pieces of the application too to make it easy for me to recognize them.

1. Installations / Software needed

Download and install –
We will utilize the Selenium IDE, Mozilla Firefox (any version as long as each is compatible with the other)
Mozilla Firefox - http://www.mozilla.org/en-US/firefox/new/
Selenium IDE - http://seleniumhq.org/download/ (download Selenium IDE latest version and any add-ons listed in the same page (optional))
FireBug for Firefox – http://getfirebug.com/downloads

2. Create folders –

Select a location on the local Drive and create the following folders in the order specified. The folder names that are highlighted (In **Bold**) only will be used for this tutorial.



000_SeleniumIDE
- 000_Archive
- **001_TestResults**
  - **07_Login_Gmail**
  - **07_Login_Netflix**
  - **07_Login_Yahoo**
- **001_TestSuites**
- **01_FilePath**
- **02_Repository**
- **03_DataSheets**
- **04_GoToURL**
- 05_BVT
- 06_Register
- 06_ForgotPassword
- 06_ForgotUsername
- 06_Other
- **07_Login**

Figure 1 – Create Folders

Invoke Mozilla Firefox. Go to menu – Tools > Selenium IDE. Selenium IDE comes up in record mode. Stop recording (click on the “pink” button on the top right side. Delete any lines of code that may have been recorded.

3. 01_FilePath

File path specifies the location for the Captured Screen Shots to be saved. Each set of screen shots will be saved to a specific location with a specific naming convention.

3.1 File Path for Gmail

Click in the Source area to highlight the first line. In the below fields, type in –

Command = store
Target = C:\000_SeleniumIDE\001_TestResults\
Value = first

Command	Target	Value
Store	C:\000_SeleniumIDE\001_TestResults	First
Store	07_Login_Gmail\	Second
Store	pGmail_	Third
Store	${first}${second}${third}	FilePath

Save the test case as "01_FilePath_Login_Gmail" at "C:\000_SeleniumIDE\01_FilePath"

3.2 File Path for Netflix

Copy all the lines of code from file "01_FilePath_Login_Gmail" and paste it into a new file (In Selenium IDE section Test Case, Right click and select "New Test Case".) Change the code as such –

Command	Target	Value
Store	C:\000_SeleniumIDE\001_TestResults	First
Store	07_Login_ Netflix \	Second
Store	pNetflix _	Third
Store	${first}${second}${third}	FilePath

Save the test case as "01_FilePath_Login_Netflix" at "C:\000_SeleniumIDE\01_FilePath"

3.3 File Path for Yahoo

Copy all the lines of code from file "01_FilePath_Login_Gmail" and paste it into a new file (In Selenium IDE section Test Case, Right click and select "New Test Case".) Change the code as such –

Command	Target	Value
Store	C:\000_SeleniumIDE\001_TestResults	First
Store	07_Login_ Yahoo \	Second
Store	pYahoo _	Third
Store	${first}${second}${third}	FilePath

Save the test case as "01_FilePath_Login_Yahoo" at "C:\000_SeleniumIDE\01_FilePath"

4. 02_Repository

Even though Selenium IDE does not offer an object repository (like in QTP) or object map (like in RFT) to store all the elements with their desired properties / attributes, we can “store” the elements by their HTML ID and / or Document Object Models with specific names (variables) that are easy to remember and relate to, when writing code in several test cases.

4.1 Store elements for Gmail

I will now capture the elements that are necessary to verify a successful or failed login for page Gmail Login.

In Selenium IDE section - Test Case, right click and select “New Test Case”.
In Mozilla Firefox, navigate to Gmail login page (Gmail).

Right click on the field Username and select option “`assertText id=Email`”
Right click on the field Password and select option “`assertText id=Passwd`”
Right click on the checkbox Stay Signed in and select option “assertText id=PersistentCookie”
Right click on the button SignIn and select option “`assertText id=signIn`”
Right click on the link Cant Access Your Account and select option “`asertText id=link-forgot-passwd`”

Log in to Gmail with valid credentials.
Right click the email ID in the Gmail Logged in page and select option “`assertText id=gbgs4dn`”
Right click on the button Log out and select option “`assertText id=gb_71`”

Log in to Gmail with invalid credentials.
Right click on the error messages in red and select option “`assertText`“

Command	Target	Value
assertText	id=Email	
assertText	id=Passwd	
assertText	id=signIn	
assertText	id=PersistentCookie	
assertText	id=link-forgot-passwd	
assertText	id=gbgs4dn	
assertText	id=gb_71	
assertText	Enter your email address.	
assertText	Enter your password.	
assertText	exact:The username or passwor...	
assertText	link=exact:?	

Figure 2 – 02_repGmail file before converting to a repository file

Save the test case as “02_repGmail” at “C:\000_SeleniumIDE\02_Repository”.

4.1.1 Store

In Windows Explorer and go to location – "C:\000_SeleniumIDE\02_Repository". Open file "02_repGmail" in Notepad (or WordPad also suffices). Replace text "`assertText`" and "`assertTitle`" with "`store`" via find and replace (Ctrl + h). Replace anything in column – Command to "`store`".

4.1.2 English names

Open the file again via Selenium IDE. To the HTML ID of field username (`id=Email`), specify Value as "`fUsername`", meaning – f = field, Username = username. I find it easy to remember the elements I use for coding when I name them in English as they would appear on the webpage along with subtly specifying the element type.

For more operational naming, I usually specify the page name too like – `pGmailLogin_fUsername`, meaning – p = `page`, `GmailLogin` or `GL` (to keep the names short while making sense) = Gmail Login, f = field, Username = username. There is no right or wrong way to name the elements as long as they make sense to the person coding the test cases.

4.1.3 Naming convention

```
p = Page,
f = Field
b = Button
ddl = Drop Down List
cb = Check Box
rb = Radio Button
l = Link
t = text
d = Dialog Box
```

```
logo = Logo
img = Image
m = Menu
sm = Sub Menu
sec = Section of a page
err = Error
tab = Tab
cap = Caption
```

These are the most commonly used shortcuts by me to specify the element type. I habitually abbreviate the page names but only occasionally I abbreviate the element names.

The names are short or long based on the hierarchy. Example for short names: "`pGL`" is short and meaningful enough for our test cases. If we also were dealing with page General Ledger, I would name the page – `pGenLed` to be able to differentiate easily. Example for long names: "`pGL_dLoginFailure_bOK`." Or "`pGenLed_dLogFail_errUsernameAndPasswordNoMatch`"

Name the rest of the elements / attributes as discussed.

4.1.4 Sort Order

Sort the column = Value alphabetically. This test case file will acquire many additional elements each day. It would be difficult to keep track of the elements when there is sudden need to change or add or even look for, if particular elements already exist. So, sort all the lines in alphabetical order in the Value column to look like the below picture. A little effort will save a lot of time and sanity in the long run.

Command	Target	Value
store	id=signIn	bSignIn
store	id=gb_71	bSignOut
store	id=PersistentCookie	cbStaySignedIn
store	id=Passwd	fPassword
store	id=Email	fUsername
store	link=exact:?	imgWhat
store	id=link-forgot-passwd	lCantAccessAccount
store	id=gbgs4dn	tUsername
store	Enter your email address.	tUsername_Error
store	Enter your password.	tPassword_Error
store	id=errormsg_0_Passwd	tUsernamePassword_NoMatch

Figure 3 – 02_repGmail file after sorting the given names

4.2 Store elements for Netflix

In Selenium IDE section Test Case, Right click and select "New Test Case".
In Mozilla Firefox, navigate to Netflix login page (Netflix).
Capture all the elements on the page. Log in with valid credentials and add the page title and link / button Logout.
Save the test case as "02_rep_Netflix_HomePage" at "C:\000_SeleniumIDE\02_Repository". Following the steps discussed in 'Store elements for Gmail', name the attributes and sort.

Command	Target	Value
store	id=login-form-contBtn	bSignIn
store	link=Sign Out	bSignOut
store	//form[@id='login-form']/di...	cbStaySignedIn
store	id=password	fPassword
store	id=email	fUsername
store	xpath=(//a[contains(text(),'Cl...	lCantAccessAccount
store	link=Click here.	lRegister
store	//form[@id='login-form']/di...	lWhatsThis
store	link=Sridevi Balla	tUsername
store	css=#aerrors > ul	tUsernamePassword_NoMatch

Figure 4 – 02_repNetflix file after sorting the given names

4.3 Store elements for Yahoo

In Selenium IDE section Test Case, Right click and select "New Test Case".
In Mozilla Firefox, navigate to Yahoo login page (Yahoo).
Capture all the elements on the page. Log in with valid credentials and add the page title and link / button Logout.

Save the test case as "02_rep_Netflix_HomePage" at "C:\000_SeleniumIDE\02_Repository". Following the steps discussed in 'Store elements for Gmail', name the attributes and sort.

Command	Target	Value
store	id=signUpBtn	bCreateAccount
store	id=.save	bSignIn
store	id=yui_3_2_0_7_134025709554...	bSignOut
store	id=persistent	cbStaySignedIn
store	id=passwd	fPassword
store	id=username	fUsername
store	id=.save	lCantAccessAccount
store	css=span.yuhead-name	tUsername
store	css=div.yreqertxt	tUsernamePassword_NoMatch

Figure 5 – 02_repYahoo file after sorting the given names

Note: I did not specify the '`pGmail_`' or '`pYahoo_`' or '`pNextflix_`' for any of the attributes because, I am demonstrating the reuse of scripts with multiple yet similar user interfaces. My objective here is to show how script death can be minimized when the user interface changes.

5. Documentation (optional)

Take some time to document the elements in any kind of file that you would like. I prefer to document the naming convention of each application and its pages in Excel spreadsheets. Specify values to the rest of the attributes too using the suggestions discussed. The end result would look somewhat like the picture below.

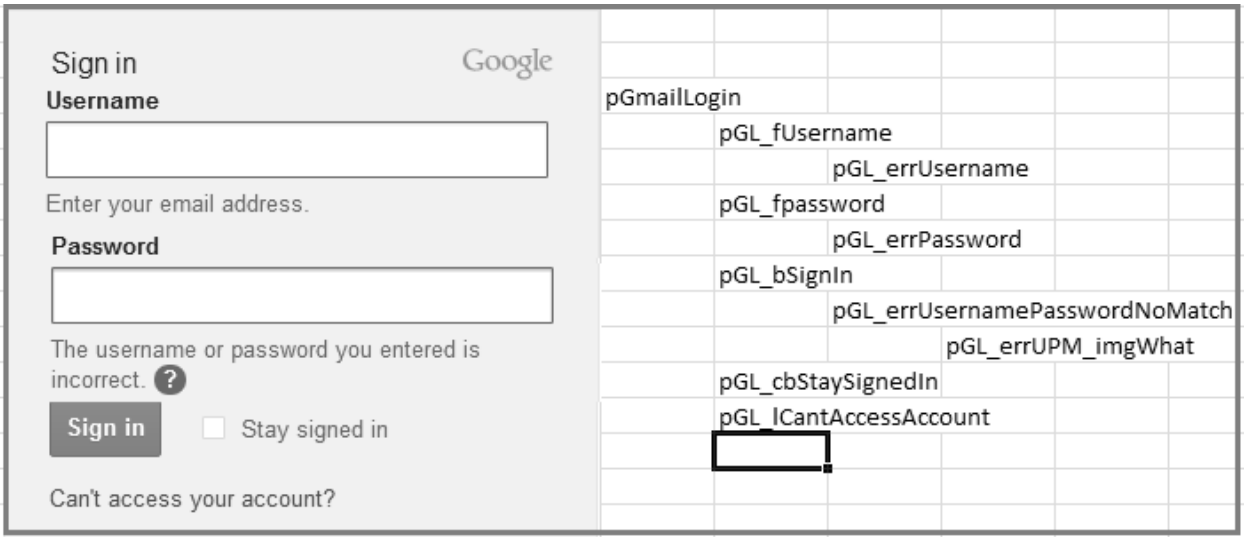


Figure 6 – 02_repGmail file after converting to a repository file

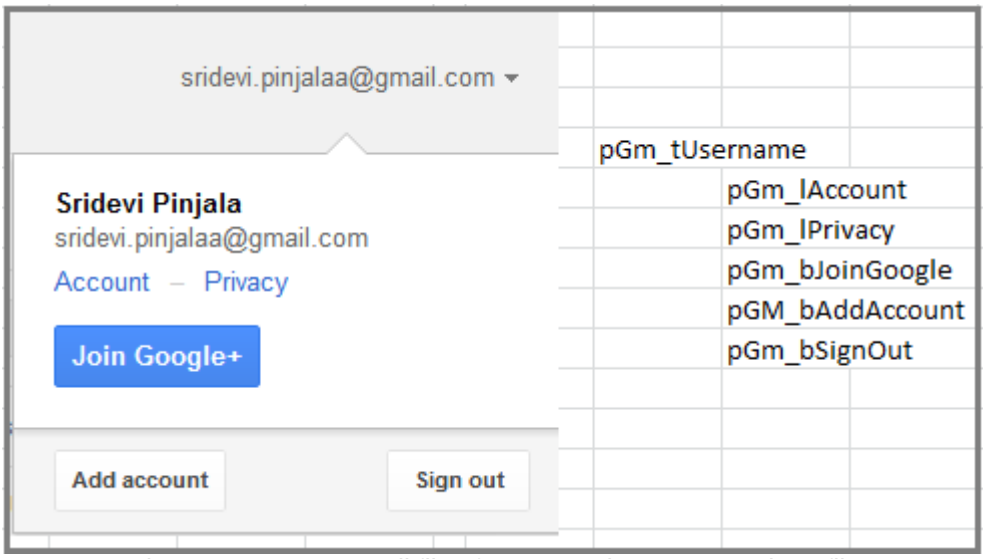


Figure 7 – 02_repGmail file after converting to a repository file

The span of the name does not matter, its uniqueness matters. The purpose for naming is to recall the element and utilize them promptly. Meaningful names will also eliminate the possibility of naming two attributes the same. Maintaining the given names in the hierarchical order in the Excel spreadsheet, gives me clarity about the requisite elements needed to be tested before utilizing / testing an element. (Example: Unless the page Gmail is invoked, the login test case cannot be executed)

5.1 More about 02_Repository

5.1.1 02_rep(page)

Create a repository file (page) to store all the elements per page. Some pages have just 2 elements specific to the page. Some pages have 200 elements specific to the page. I store the page specific elements for each page in an application in an individual repository file. This way when the User Interface or Document Object Model or HTML IDs of the page change, it would be easy to locate the particular page and update the values quickly.

5.1.2 02_repHomeElements

Create a repository file home to store all the home elements. I call the elements (links, logos, version, header, footer, menus etc.) that are common on all the pages of a web application as the Home elements. This practice eliminates the need to maintain the standard elements in all the individual page repository files and also eliminates the need to update numerous files.

5.1.3 02_repPages

Create a repository file Pages to store all the page titles. Most pages may not need individual repository files, but it would be necessary still test their existence. Though the elements of each page are

maintained in individual files, it is a good practice to store all the page titles in a singular file. It would be easier to update the Titles in one quick editing session.

5.1.4 Create 02_repPages

Go to home pages for Gmail, Netflix login and Yahoo mail. Right click anywhere on the page. Select option "`assertTitle`". Add the variable name in the column Value as such - for Gmail add `pGmailLogin`, for Netflix login add `pNetflixLogin`, for Yahoo Mail add `pYahooLogin`.

Note: We have successfully created the pages repository file necessary for our test today. In real time, the pages of a particular application only are to be added to this file.

6. 03_DataSheets

Selenium IDE does not support data driven testing. However, we can create files specifically to store data. Several of these files can be created ahead of time and saved for iteration. This kind of data files may also be used for setting up data, to keep track of the data already used and so on. Instead of in an Excel sheet, the data is maintained in the Selenium test case itself.

Each data using test case does not need an individual data file. For example: the file used only once for Registration can also be used for several Logins and profile changes, etc. (For now, only data for Username and Password will be discussed.)

6.1 Data Sheets for Gmail

In Selenium IDE section Test Case, Right click and select "New Test Case".
Click in the Source area to highlight the first line. In the below fields, type in –

Command	**Target**	**Value**
Store	GmailUsername	data_Username
Store	Abc123^^	data_Password

Save the test case as "03_data_Gmail_Register" at "C:\000_SeleniumIDE\03_DataSheets".

6.2 Data Sheets for Netflix

In Selenium IDE section Test Case, Right click and select "New Test Case".
Copy all the lines of code from file "01_data _Gmail_Register". Modify data as necessary.

Command	**Target**	**Value**
Store	NetflixUsername	data_Username
Store	Abc123^^	data_Password

Save the test case as "03_data_Netflix_Register" at "C:\000_SeleniumIDE\03_DataSheets".

6.3 Data Sheets for Yahoo

In Selenium IDE section Test Case, Right click and select “New Test Case”.
Copy all the lines of code from file “01_data _Gmail_Register”. Modify data as necessary.

Command	**Target**	**Value**
Store	YahooUsername	data_Username
Store	Abc123^^	data_Password

Save the test case as “03_data_Yahoo_Register” at “C:\000_SeleniumIDE\03_DataSheets”.

6.3.1 About data sheets

I like to name the data variables as “`data_(name)`”. This way the data variables won’t interfere with repository variables. I would also highly recommend maintaining a main data file for each set of functionalities. This main data sheet could be cloned and reused with a new name and updated data. Uniquely naming the data files will help identify the data we seek for a particular test results.

7. 04_GoToURL

Selenium typically records the default URL each test case is supposed to invoke before executing the steps in it. There are plenty of occasions where we would like to use the steps created on one environment in another (example: login script created in environment QA1 is to be used on the build installed in environment QA2). There are plenty of occasions where we would like to continue running various test case steps on a particular environment too. For this purpose, I would highly recommend maintaining a file specifically for invoking the URL alone.

7.1 04_GoToURL for Gmail

In Selenium IDE section Test Case, Right click and select “New Test Case”.
Start recording (click on the red button. The red button turns pink.)
In Mozilla Firefox, navigate to Gmail login page.
Stop recording.
Add (write) steps to verify title and to capture a screen shot of the page for reference.

Command	**Target**	**Value**
Open	\	
assertTitle	${pGmailLogin}	
captureEntirePageScreenshot	${FilePath}.png	

Save the test case as “04_GoTo_Gmail” at “C:\000_SeleniumIDE\04_GoToURL”.

7.2 04_GoToURL for Netflix

In Selenium IDE section Test Case, Right click and select “New Test Case”.
Copy all the steps from test case “04_GoTo_Gmail” and paste them in the new Test Case.

Update "${pGmailLogin}" to "${pNetflixLogin}"
Save the test case as "04_GoTo_Netflix" at "C:\000_SeleniumIDE\04_GoToURL".
Via Windows Explorer navigate to the location "C:\000_SeleniumIDE\04_GoToURL".
Open the file "04_GoTo_Netflix"
Replace the URL Gmail login page with Netflix login page.
Save the file and close it.

7.3 04_GoToURL for Yahoo

In Selenium IDE section Test Case, Right click and select "New Test Case".
Copy all the steps from test case "04_GoTo_Gmail" and paste them in the new Test Case.
Update "${pGmailLogin}" to "${pYahooLogin}"
Save the test case as "04_GoTo_Yahoo" at "C:\000_SeleniumIDE\04_GoToURL".
Via Windows Explorer navigate to the location "C:\000_SeleniumIDE\04_GoToURL".
Open the file "04_GoTo_Yahoo"
Replace the URL Gmail login page with Yahoo login page.
Save the file and close it.

8. 07_Login scripts for Gmail, Netflix, Yahoo

We are now going to create selenium test cases for Login Steps, Login Pass, Login Fail and Log Out. These test cases will be hand written (keyboard typed), not recorded using the variables created in the previous files. These test cases can be used for all the three user interfaces (Gmail, Netflix, and Yahoo).

In Selenium IDE section Test Case, Right click and select "New Test Case".
Manually type the following in Command, Target, and Value.

8.1 Login Steps

Command	**Target**	**Value**
Type	${fUsername}	${data_Username}
type	${fPassword}	${data_Password}
clickAndWait	${bSignIn}	

Save the test case as "07_LoginSteps" at "C:\000_SeleniumIDE\07_Login".

8.1.1 Purpose of Login Steps

Type in field Username the data specified for the Username;
Type in field Password the data specified for the Password;
Click button Sign In and wait for the page to load.
Note: This test case only carries out the steps for a test. It does not verify if the test had passed or failed.

8.2 Login Pass

Command	Target	Value
assertElementPresent	${tUsername}	
captureEntirePageScreenshot	${FilePath}${Username}_PASS.png	

Save the test case as “07_LoginPASS” at “C:\000_SeleniumIDE\07_Login”.

8.2.1 Purpose of Login Pass

Only if text Username is present execute the rest of the steps;
Capture and save a screen shot at the specified location as PASS.
Note: This test case executes and logs a picture as PASS, if the desired outcome (text Username) exists.

8.3 Login Fail

Command	Target	Value
assertElementNotPresent	${tUsername}	
assertElementPresent	${fUsername}	
captureEntirePageScreenshot	${FilePath}${Username}_FAIL.png	

Save the test case as “07_LoginFAIL” at “C:\000_SeleniumIDE\07_Login”.

8.3.1 Purpose of Login Fail

Only if text Username is NOT present execute the rest of the steps;
Only if field Username is present execute the rest of the steps;
Capture and save a screen shot at the specified location as FAIL.
Note: This test case executes and logs a picture as FAIL, if the desired outcomes (text Username does not and field Username) exist.

8.4 Log Out

Command	Target	Value
assertElementPresent	${tUsername}	
Click	${tUsername}	
clickAndWait	${bSignOut}	
captureEntirePageScreenshot	${FilePath}${Username}_LogOut.png	

Save the test case as “07_LogOut” at “C:\000_SeleniumIDE\07_Login”.

8.4.1 Purpose of Log Out

Only if text Username is present execute the rest of the steps;
Click the text Username;
Click the button Sign Out and wait for the page to load.
Capture and save a screen shot at the specified location as LogOut.
Note: This test case executes and logs a picture as LogOut only if the desired outcome (text Username) exists.

9. Base Script

For effortless and quick test case creation, I would recommend creating and maintaining a base script. The base script will consist of all the elements needed for writing a test case for most actions, verifications and assertions.

Command	**Target**	**Value**
assertTitle	${p}	
assertElementPresent	${}	
assertElementNOTPresent	${}	
click	${rb}	
click	${cb}	
click	${l}	
click	${b}	
type	${f}	${data_}
select	${ddl}	${data_}
captureEntirePageScreenshot	${FilePath}${}_.png	

10. Test set arrangement and execution order

We have created test cases in modular fashion. It is time to put them together sensibly. As I had already mentioned,
01_FilePath (files – 01_FilePath_First, 01_FilePath_Second, 01_FilePath_Third),
02_Repository (files – 02_repPages, 02_repHomeElements files; If specific page or pages are involved in the test, 02_rep (page(s)) will also be necessary in the test suit),
03_DataSheet (optional, datasheets may be added as necessary or may be skipped for some test sets),
04_GoToURL (one file suffices)
The test cases should be added in ascending order till 04_GoToURL.

This is one way to create, store and guesstimate when a test case is ready and if it is set up in order. Now, the actual test cases may be added in the order they would execute manually.

11. Test Execution and Results

Typically, all test cases are supposed to turn Green in order to assume that the test set passed. Here, that is not the case. Certain test cases are supposed to pass and certain test cases are supposed to fail. Each test case will only execute if the required conditions are met.

Provide valid credentials to pass a test. Passed tests would have three picture logs, one for navigating to the website, one for successful login, one for log out.

Provide invalid credentials to fail a test. Failed tests would have two picture logs, one for navigating to the website, one for failed login.

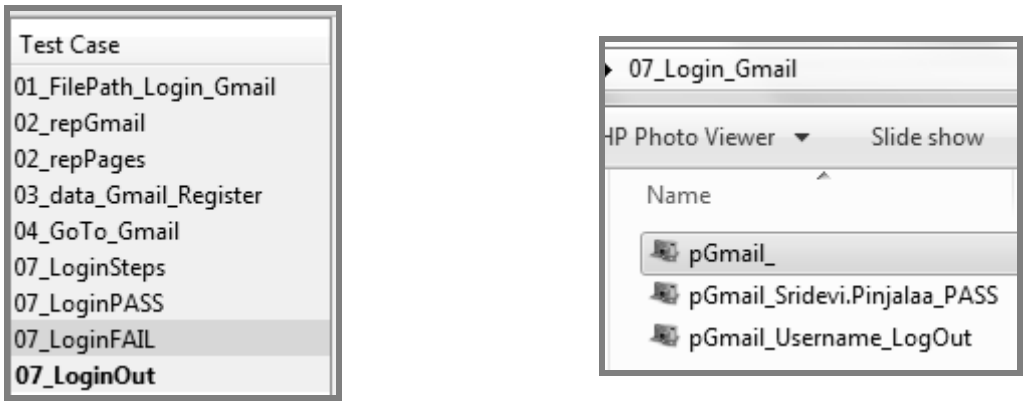


Figure 8 – Gmail Login – PASS



Figure 9 – Gmail Login – FAIL

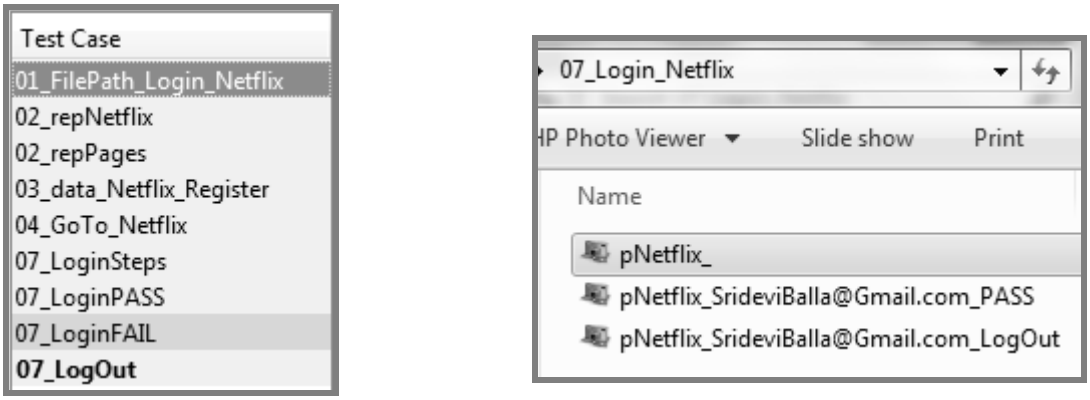


Figure 10 – Netflix Login – PASS

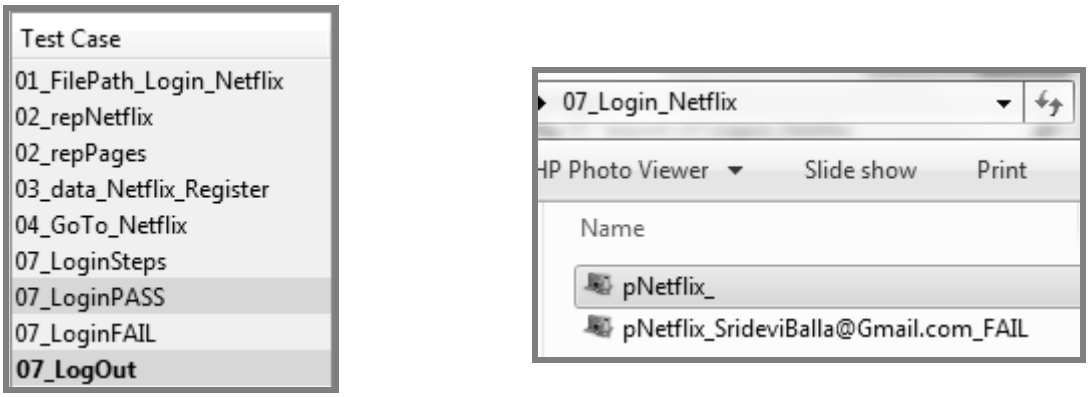


Figure 11 – Netflix Login – FAIL

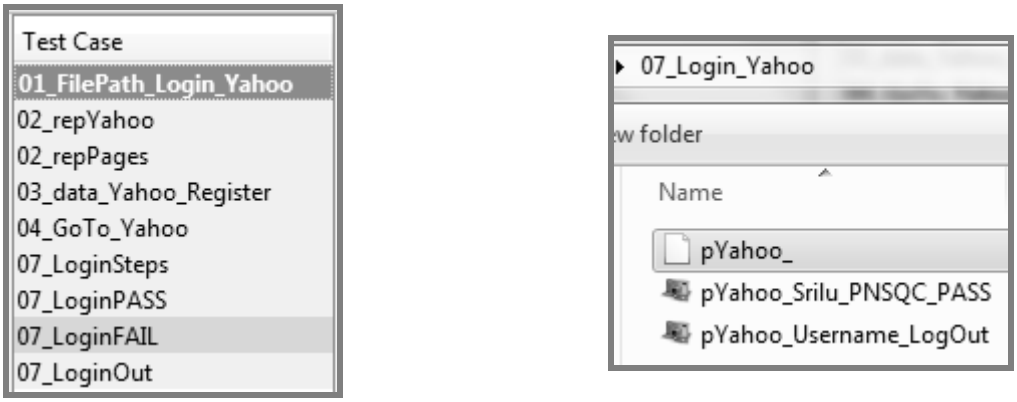


Figure 12 – Yahoo Login – PASS



Figure 13 – Yahoo Login – FAIL

12. SAREE

The most suitable mnemonic to summon up the steps for Green Lantern Automation concept would be "Select, Assert, wRite, Execute, Exact" (SAREE). The scripts, just like a Saree can be wrapped to any User Interface testing as long as their functionalities are similar.

Select elements for testing.

Select the elements that would be used for testing. Listed below are a few common elements.

- Page
- Field (text field)
- Button
- Check box
- Radio button
- Link
- Menu
- Sub Menu
- Drop Down List
- Caption
- Image
- Logo
- Text
- Section
- Dialog box

Assert appropriate custom name to each element in the repositories

Rename a few of the above elements to describe them fittingly. Listed below are a few elements named to fit Google page.

- Page → `pGoogle`
- Field (text field) → `pGoogle_fUsername`
- Button → `pGoogle_bSearch`
- Check box → `pGoogle_cbChose`
- Radio button → `pGoogle_rbTrue`
- Link → `pGoogle_lHome`
- Menus → `pGoogle_mMore`
- Sub Menu → `pGoogle_smTranslate`
- Drop Down List → `pGoogle_ddlSettings`
- Caption → `pGoogle_capGoogle`
- Image → `pGoogle_imgGmail`
- Logo → `pGoogle_logoGoogle`
- Text → `pGoogle_tVersion`
- Section → `pGoogle_secSearch`
- Dialog box → `pGoogle_dSuccessful`

Based on the naming convention one can (with a little bit of training) can identify that the elements belong to the page Google

wRite the test cases with custom names

Selenium IDE typical code would look as such **–**

Command	**Target**	**Value**
`type`	`${pGmail_fUsername}`	`${data_Username}`
`click`	`${pGoogle_bSearch}`	
`select`	`${pGmail_ddlEnvironment}`	`${data_Environment}`
`assertTitle`	`${pGoogle}`	

Execute the test cases in test sets.

Exact the element property in the repositories from time to time

13. Conclusion

Most of the time automated scripts are created by record and play method. This method only imitates the user's actions. This method won't log warning, error messages or perform comparisons that are particular to a function or application. Automated scripts and suits created for a particular application cannot be reused with other similar applications, especially if the UI changes are drastic. The effort and time that goes into the development of automated scripts will go waste as soon as changes are made to the application.

With the Green Lantern Framework, the scripts are safe. Major or minor UI changes to an application will not cause script death. The test cases created for one application can also be used for applications with similar functionality. Since the test cases are reusable (green), the engineers will have a lot of time to focus on creating new test scenarios versus constantly updating the same test cases.

Since the UI elements are stored in a 'repository' file, the framework enables engineers to create test cases even before an application build is released for testing. When the build is ready, the tester can populate the repository file(s).

Green Lantern Framework - select elements for the test, assert custom names in English as they appear on the interface, wRite the test cases, execute the test cases in test sets and exact the element ID or Name in the repository file.

Works Cited

Gmail. (n.d.). Retrieved from https://accounts.google.com

Green Lantern. (n.d.). From http://en.wikipedia.org/wiki/Green_Lantern

Netflix. (n.d.). Retrieved from https://signup.netflix.com/Login

SAREE. (n.d.). From http://en.wikipedia.org/wiki/Sari

Yahoo. (n.d.). Retrieved from https://login.yahoo.com

Cost Effective Agile Test Practices and Test Automation using Open Source Tools SpecFlow and White

Kavitha Naveen and Perry Hunter
Tektronix
kavin@tek.com and perry.w.hunter@tektronix.com

Abstract

Budget! Cost optimization! Expensive! These are common words which are often heard when licensed automation tools are proposed for software test automation.

If licensed test tools are expensive and budget is the key constraint, don't worry; Open source tools are there to help us in creating effective and low-cost test automation. Open source tools have an added advantage of supporting Behavior-Driven Development (BDD) which is an agile software development technique that encourages collaboration between developers, testers and non-technical or business participants in a software project. Open source tools are free; just get it and use it to see the power of test automation.

This paper is about a Test Framework which was developed using open source tools to test plugins embedded in Visual Studio 2010.

Among the many open source tools available, this presentation will showcase how two tools, SpecFlow and the White library, have made test automation tasks easy to complete!

Writing elaborate test cases is a big pain, isn't it? Don't worry; BDD comes to your rescue. You can write simple and easily readable test cases in the language you speak. SpecFlow from TechTalk brings BDD to the .NET environment. It can be used to write test cases using BDD style in Visual Studio.

For GUI automation we have the White library from Thought Works. It supports automation for Visual Studio 2010 Controls and plugins embedded in Visual Studio 2010.

Biography

Kavitha Naveen is a senior software quality lead engineer at Tektronix Engineering Development India Ltd. Over the past few years, she has been involved in test automation for products used in Test and Measurement Industry. Kavitha has worked extensively on training test engineers and researching test methodologies. Kavitha has a Bachelor's of Engineering in Telecommunication from Bangalore University, India.

Perry Hunter is a senior software quality lead engineer at Tektronix in Beaverton, OR. He has worked in the fields of scientific research and software quality for approximately 25 years, and is currently supporting the signal generator group at Tektronix. He enjoys gardening, most products of fermentation and is passionate about Kendo. His first real computer was a Data General MV/8000.
Perry holds a Bachelors of Science in Oceanography from Humboldt State University.

1 Introduction

Every software test engineer will be required to write test cases at some point during the software project cycle to test the product under development. Each person follows his own method to write the test cases. Some test engineers write elaborate test cases while others write very brief test cases. Finally, when the test cases are ready, they need to be reviewed. Usually, when the test cases are sent for review there are hardly any review comments, as the software team finds reasons not to do it. Even if it is done, the review is often not very extensive. The end result is frequently test cases not getting reviewed and missing test conditions— which are defects seen later in the end product.

One of the reason for test cases not getting reviewed is they are frankly less interesting than coding to the team members. No one likes to switch context and decipher elaborate test code. Therefore if we write test cases in a more readable language, use precise words to explain the test scenario and then automate the test case line by line, then the test case review time can be substantially reduced. The quality of the test cases is improved, because the reviewers can clearly understand what is being tested and how.

In this paper, we describe how two open source tools; i.e., SpecFlow and White, combined with Microsoft Visual Studio, help us to carry out GUI test automation using agile test practices in the Windows environment. SpecFlow helps us to write test cases using a BDD style in the .NET environment with simple sentences which are easily understood by all stakeholders. The White Library from Thought Works helps us to carry out GUI test automation.

We will also share how we automated Microsoft development tools like Visual studio 2010 and achieved nearly 90% automation for one of our projects using the open source tools. The whole process—starting from evaluating the tools, implementing the automation scripts, and running the scripts successfully for the project—took about 12 weeks' time.

2 Incorporating Agile Testing using Open Source Tools

Agile testing has become a common phrase in the software industry. But now the question is, where do I start and how do I proceed? When it comes to following agile test practices, the concentration is more on team work rather than individuals like testers or developers. Hence, our thought process also needs to change. Test engineers need to think of following agile practices right from the start

Agile for us really means doing things in a simplified but rapid way. For writing the test cases and collaborating well with the requirements stakeholders, BDD comes to the rescue. Here the test cases are written in simple, business readable language. BDD is a widespread methodology used in the agile community.

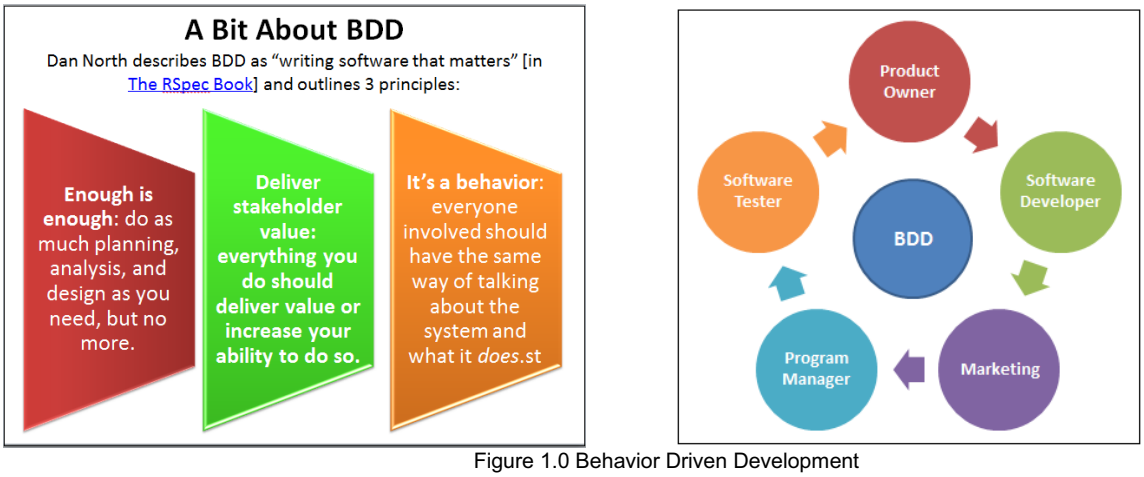


Figure 1.0 Behavior Driven Development

One of the reasons why the defects escape in the released product is due to requirements not being clear and thereby missing test cases or incorrect test cases. By using BDD to write the requirements and test cases wherein emphasis is given to the vocabulary, writing incorrect test cases reduces to a great extent.

Test cases written in BDD style reduce review time as they become automated. The test case is written in the form of Features and Scenarios.

2.1 Cucumber

Cucumber (written in the Ruby programming language) is a tool used for writing test cases in BDD style. Gherkin is the language that is used to write BDD style test cases in Cucumber.

A feature is something that your software does (or should do), and generally corresponds to a user story and a way to tell when it's finished and that it works.

The general format of a feature is:

```
Feature: <short description>
  <story>
  <scenario 1>
  ...
  <scenario n>
```

A scenario is made up of 3 sections related to the 3 types of steps:

1. ***Given***: This sets up preconditions, or context, for the scenario.
2. ***When***: This is the action, the behavior that we're focused on.
3. ***Then***: This checks results… it verifies that the right thing happened as a result of the *When* steps.

A simple example to illustrate writing Test Cases in Cucumber BDD Style would be as given below

```
Feature: Bug Reports

As an SQA Engineer
I want to search defects
So that I can keep my team informed

Scenario: Finding severe Defects
  Given there are 5 defects on "Project A"
  And 2 of those are marked "Critical"
  When I search for defects marked "Critical" on "Project A"
  Then I should see 2 defects
```

All the test cases written above are in plain English. Note the simple sentences used to describe the feature and the test case scenarios. When test cases are written in this style the entire can more easily understand the given conditions, the action conditions and the expected result conditions.

If the right tool exists to enable and simplify writing BDD style test cases and to help us follow agile test practices, then the burden on the team to follow agile practices will be drastically reduced.

Once the right tool is selected for test case development, then the other tasks like test script development and functional testing will fall in place. The team will be following Agile Test practices right from test case development.

In the next section we will see how the open source tool SpecFlow is used for writing test cases in BDD style.

3 Selecting SpecFlow for Behavior Driven Development in the .NET environment

As explained in previous section, BDD helps us to write test cases in plain English and then convert the same test cases to test scripts. In the .NET environment, SpecFlow comes in handy for writing BDD style test cases. It is an open source tool that supports the implementation of the ideas behind agile methodologies like BDD. It lets you write specifications using 100%-Cucumber-compatible Gherkin Syntax within the Visual Studio environment. In the upcoming sections, we shall see the advantages of SpecFlow.

3.1 SpecFlow is Open Source

The budget gets tightened often in organizations due to reasons like recession, low profit margins, and managers always demanding “quicker, cheaper, better” solutions. SpecFlow being an open source tool is an added advantage due to these reasons. It is a lightweight installation and integrates with Visual Studio automatically once you install the software. Using open source tools helps us lower cost of ownership and reduces dependencies on specific products. It promotes faster time to market, flexibility to customize, and ease of procurement. Another advantage is the open source community which is there to support us. There is a well-led core team who quickly responds to our queries and provides solutions for us to move forward. There is continual improvement in the tool and iterative release process with a large number of developers and test engineers contributing to the tool by reporting defects and enhancement requests. Most of the expensive commercial test tools are used by a limited number of developers and testers and hence the scope for improvement is limited when compared to open source tools which are widely used. So the quality and reliability are continuously improving in open source tools.

3.2 SpecFlow Supports Agile Testing

In agile testing we need to be as quick as possible right from writing test cases to testing the product. The test team needs to be ready to run the tests on a daily basis. Instead of writing elaborate test cases, writing test cases in BDD style helps us have acceptance test cases for the requirements. The acceptance criteria are well defined in the test case itself. When the test cases are executed, there is valid proof that all the requirements have been captured along with the acceptance criteria due to the well-defined test conditions that are specified in the test case.

In software development life cycle conventionally, the testing cycle would come at the end of the project life cycle. A linear and sequential approach which can be viewed as the waterfall model is followed i.e. after all the features of the product are implemented, testing activities would start. In this model until the final stage of the development life cycle the working software with all the features implemented is not available. Defects will get accumulated and the cost of defect fixes at the end of the project life cycle becomes very huge. The development team and testing team are under severe pressure to complete their tasks during the end of the project. The probability of the working features starting to fail is high. Side effects get introduced due to defect fixes, and the whole team will be in a state of chaos. This leads to failure of the project. Hence many companies have changed their traditional approach of following water fall model, by exploring new ways which includes agile testing. Agile testing ensures quality of the product as well as the time to market. Following agile test practices as early as possible in the development cycle saves time and money of the organization

Due to the flexible processes and the ability to continuously improve the product during the development stage, agile testing has achieved significant success in recent days.

With SpecFlow, we don't have to use another test tool to capture the test cases. The aim of agile testing to complete tasks quickly and be ready for product testing can be easily appreciated when SpecFlow is used for writing test cases and converting them to test scripts. It provides a seamless developer experience by leveraging existing infrastructure and frictionless integration into the .NET platform. SpecFlow is compatible with the Gherkin language. It allows us to write test scripts in any .NET language. We use existing unit-testing frameworks as the runtime for scenario execution.

SpecFlow consists of three parts.

- Feature file converter to produce test fixtures (integrated into Visual Studio through a single-file generator)
- Runtime framework (handles step bindings, events and tracing)
- Integration with unit test frameworks (NUnit, MSTest, xUnit.Net, MbUnit)

The flow of data in SpecFlow from test case to test script and result is explained in the following diagram.

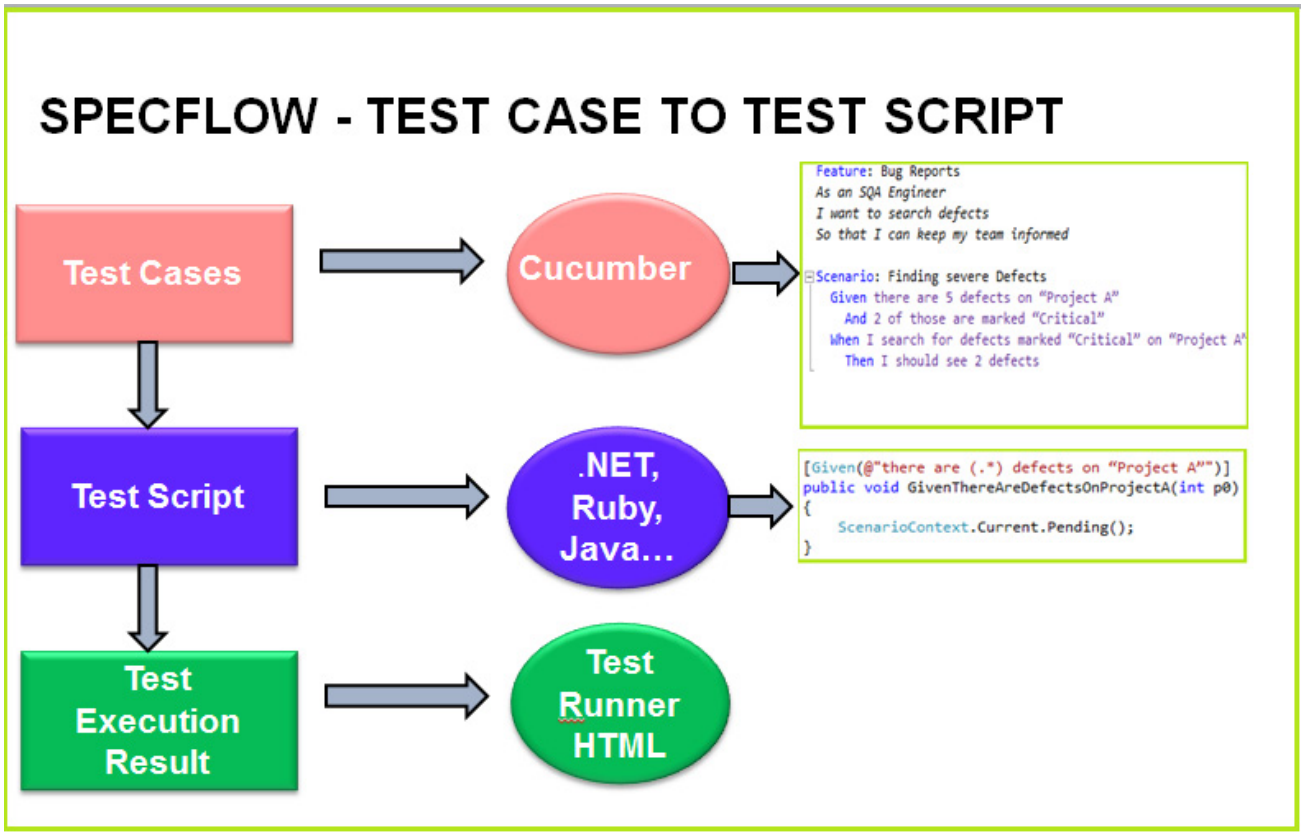


Figure 2.0 - Test Framework Architecture

3.2.1 Writing Test Cases Using SpecFlow in Microsoft Visual Studio IDE

SpecFlow is a plugin embedded in Visual Studio. SpecFlow and Visual Studio complement each other and help us to manage and automate our tests. SpecFlow helps in BDD, while the various test functionalities embedded in Visual Studio 2010 control test development and execution.

In the upcoming sections, we shall see step by step how to create test cases and execute them using SpecFlow. When SpecFlow is installed it gets added as a plugin in Visual Studio IDE as shown below.

Now we just need to add a reference to the TechTalk. SpecFlow DLL in the Visual Studio project that will contain your features.

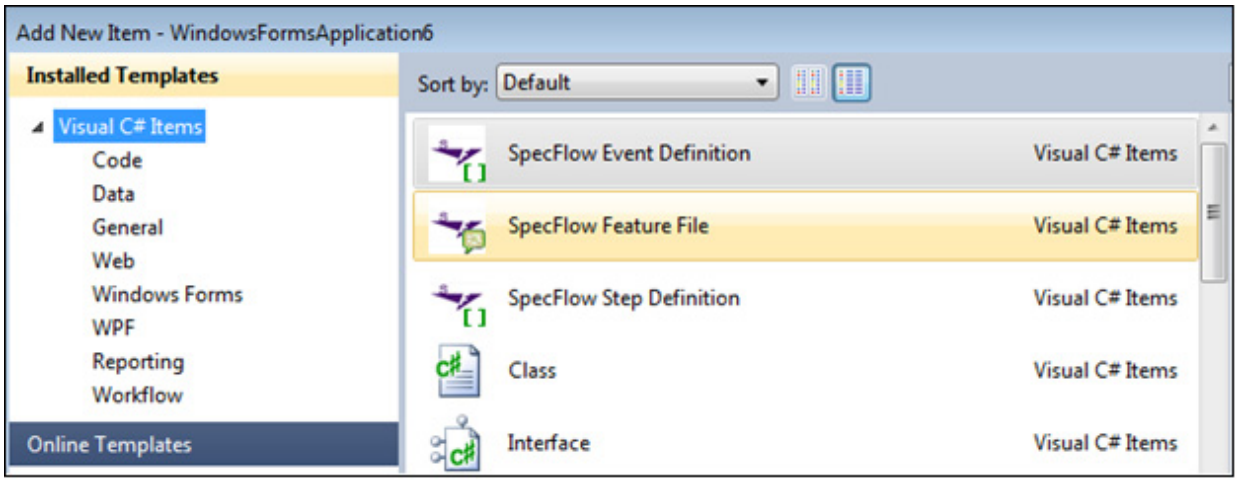


Figure 3.0 – Adding SpecFlow Feature File

3.2.2 Add SpecFlowFeature1

To write the test case, follow these steps:

- Select SpecFlow Feature File in the Installed Templates Folder as shown in the Figure 3.0
- SpecFlowFeature1.feature gets added in the project folder of Visual Studio
- A template automatically gets inserted in the SpecFlowFeature1.feature file.

Select the SpecFlowFeature1.feature file and modify the template to write the test case in BDD Style as shown in the following Figure 4.0.

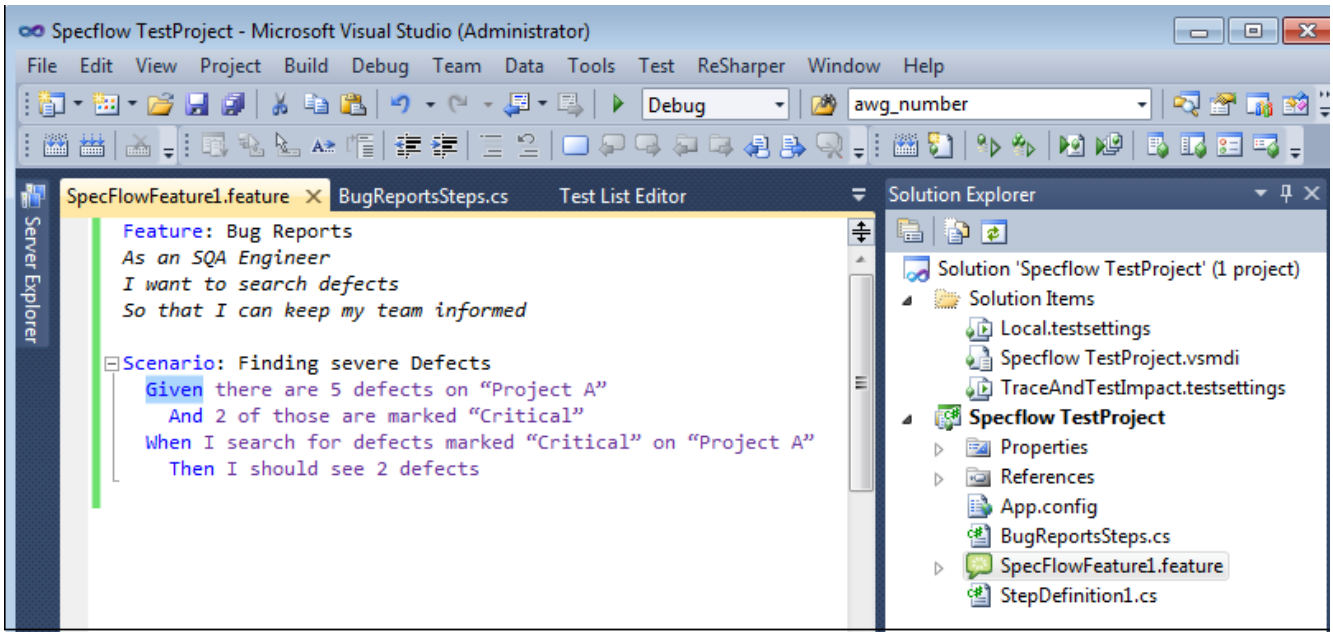


Figure 4.0 – Adding Test Cases in Feature File

3.2.3 Run Test and Watch It Fail

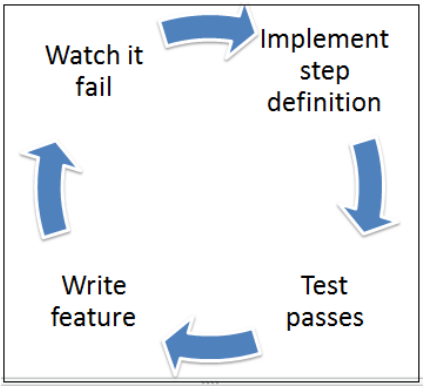


Figure 5.0 – Outside in Development in BDD

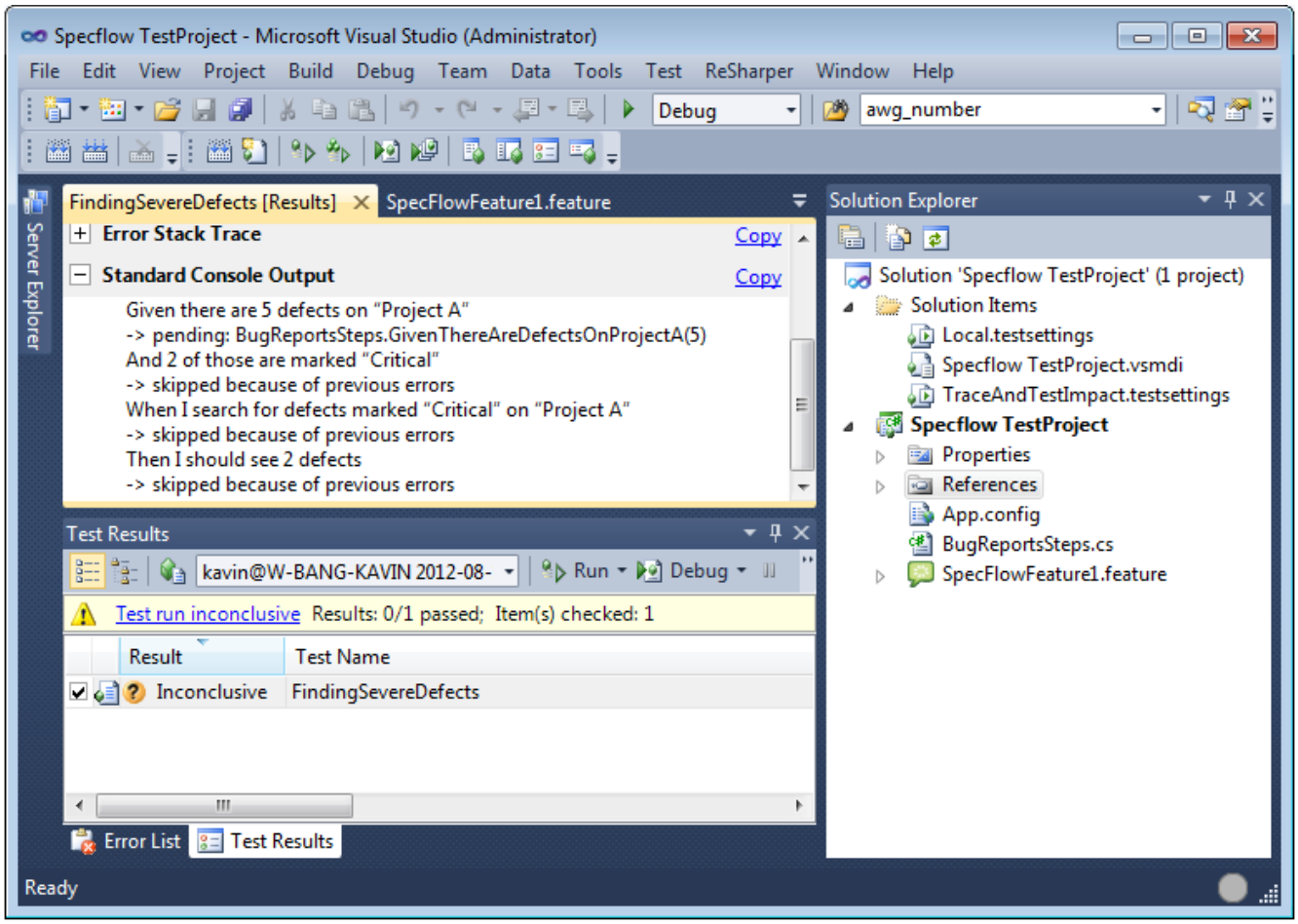


Figure 6.0 – Test Execution Results

3.2.4 Write Step Definitions

- Select SpecFlow StepDefinition file in the Installed Templates Folder
- StepDefinition1.cs gets added in the project folder of Visual Studio
- A template automatically gets inserted in the StepDefinition1.cs file

The next step would be to bind the test cases to test scripts. This would involve writing code for each of the test Steps in StepDefinition1.cs file.

For the example above, the Step Definition for the Given statement would look like the code shown in Figure 7.0

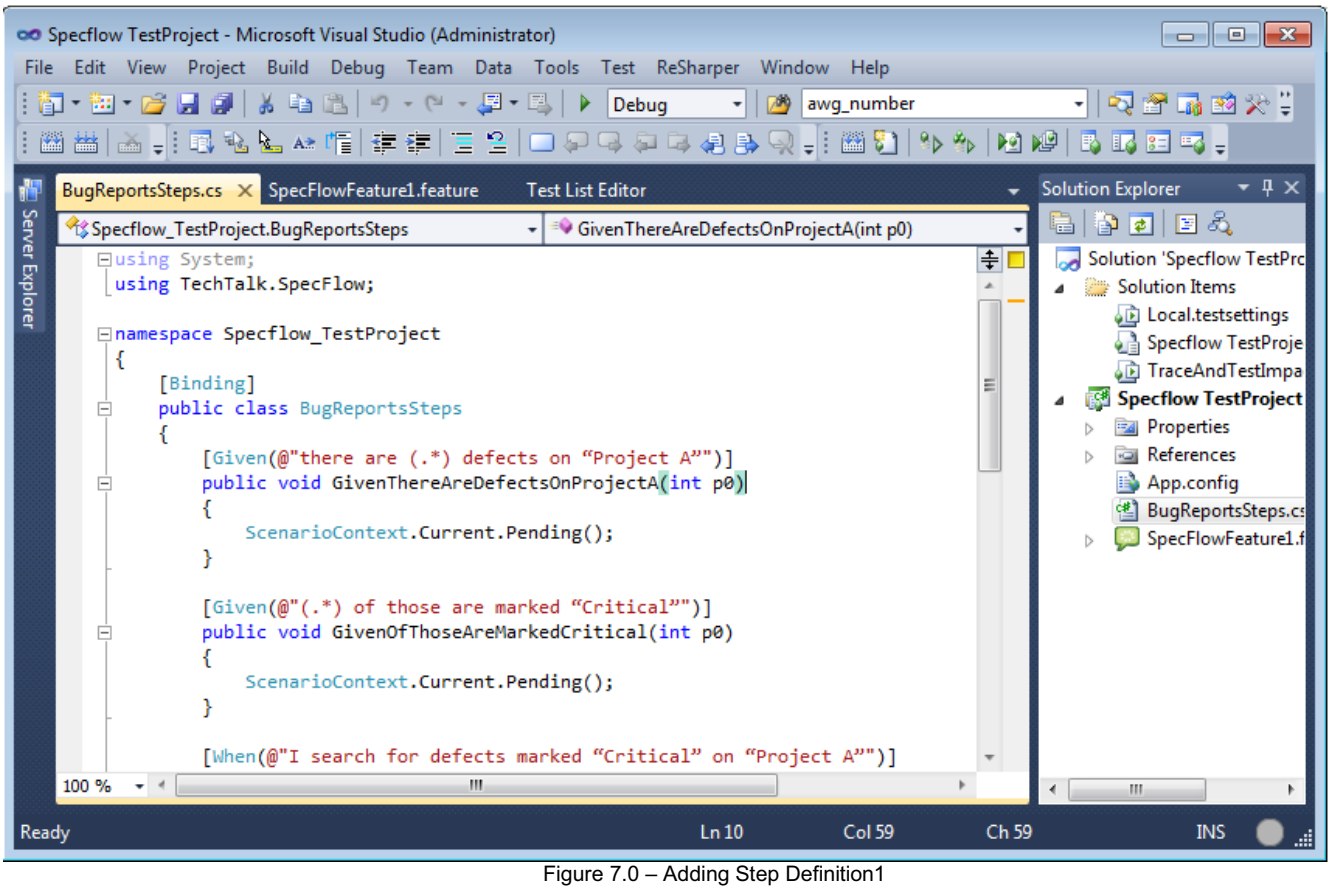


Figure 7.0 – Adding Step Definition1

Similarly, each statement in the test case needs to be bound with a step definition.

3.3 Report Format in SpecFlow

SpecFlow has an excellent report format. When you execute the SpecFlow commands from the command line, a neat reporting structure is displayed. Each line of the test case is displayed along with the test script execution time and the pass/fail results. The report format is explained in section-4.2.3

After executing the test scripts, we just need to send the report to the entire team to show the results of automation. The reporting format is very easy to read, and it serves as a useful tool to understand the results starting from test cases.

4 Selecting White for UI Automation

4.1 White is Open Source

White is another open source tool. It is based on Microsoft's automation API and provides a simple way to automate a GUI developed on Windows.

4.2 White Supports VS2010 Controls

In the Test and Measurement Industry the GUI plays a key role as it is the first item the customers get to interact with. Hence all our applications involve a GUI in some form. The primary goal for us is to ensure that the GUI works as expected in every release of the software. If we can automate even part of the GUI testing, then the work load on test engineers gets reduced.

Selecting the right tool for automation is the key element for success. In one of the applications that we work, we had added a plugin to Visual Studio 2010. The plugin enables customers to insert customized project templates in Visual Basic and Visual C#. Hence automating our plugin involved selecting the various controls in VS2010, like tool bars, menus and tree nodes, for automation. Automating the plugin involved automating Visual Studio 2010 itself. While evaluating the White library from ThoughtWorks, we found that the controls in Visual Studio 2010 are recognized and automated easily. Hence, the White library is the right choice.

Here is an example test case as shown in Figure 8.0 for Automating Visual Studio.

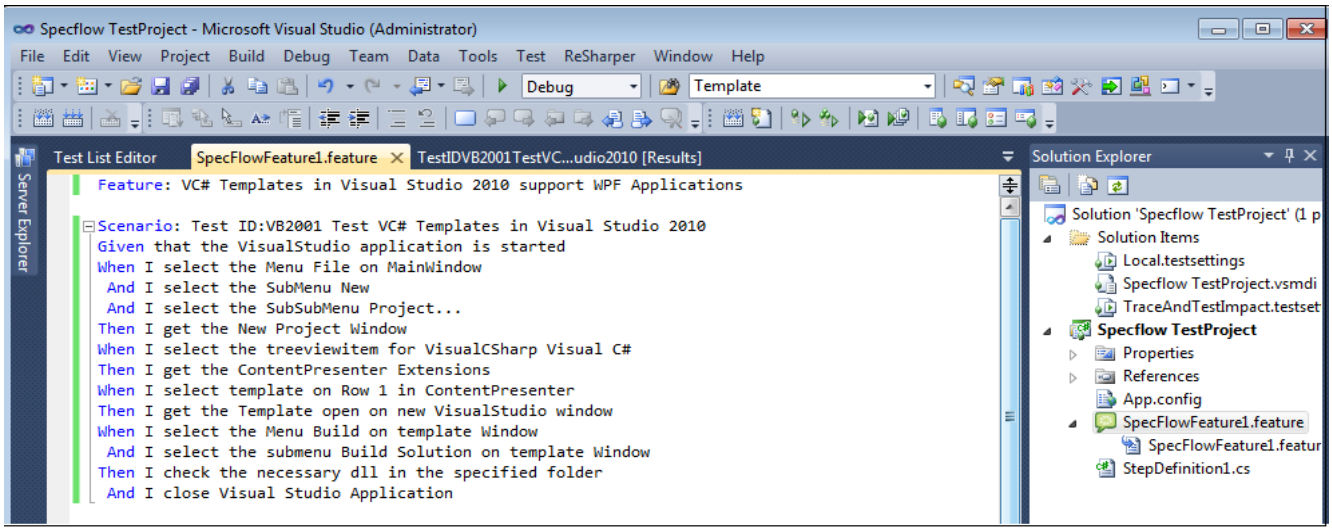


Figure 8.0 – Example Feature file for Visual Studio Automation

4.2.1 Step Definitions using White

To start the Visual Studio Application, the step definition using White functions would be like the code shown below in Figure 9.0

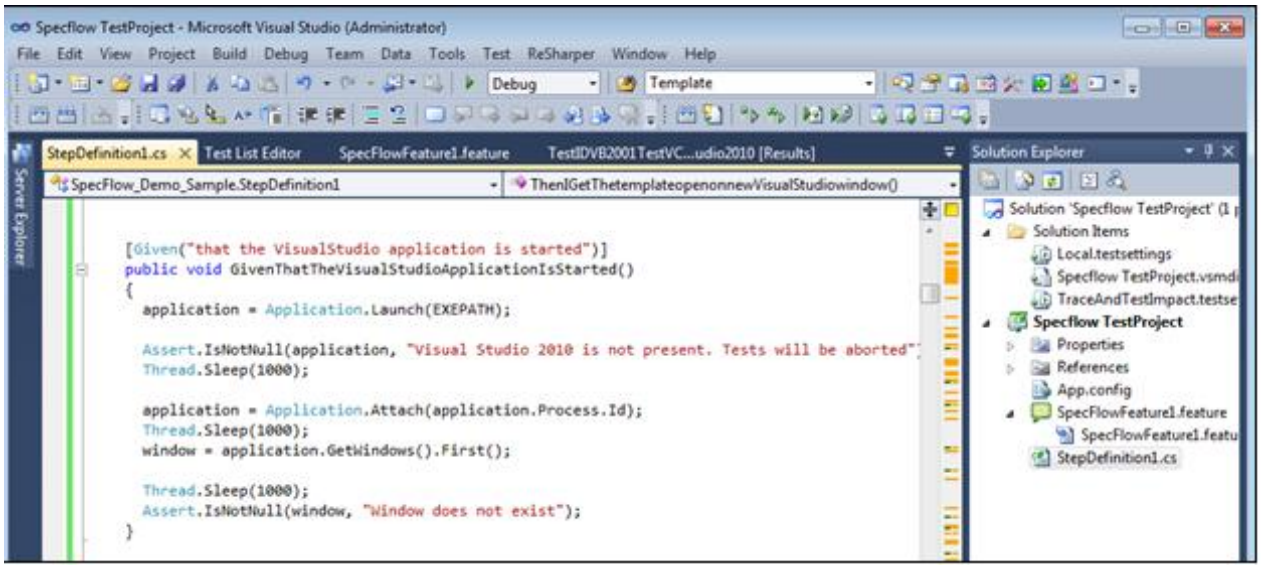


Figure 9.0 – Example Step Definition file for Visual Studio Automation

To select the File Menu in Visual Studio 2010 IDE, the step definition using White Functions would be like the code shown below in Figure 10.0

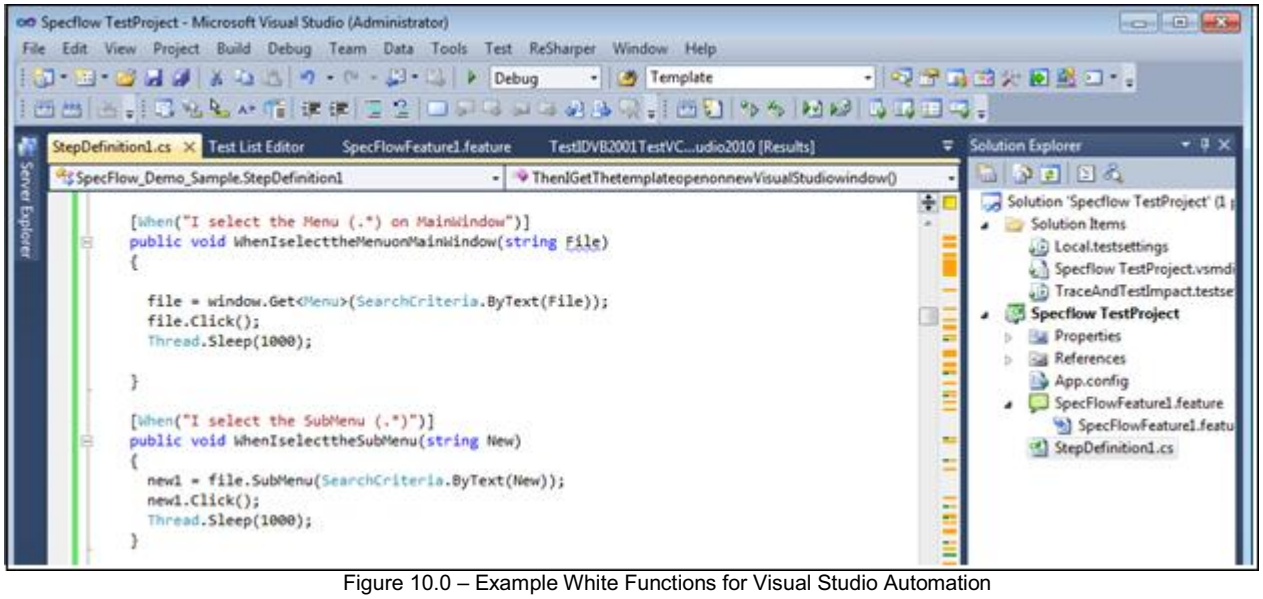


Figure 10.0 – Example White Functions for Visual Studio Automation

4.2.2 Running the Tests in the Visual Studio IDE

To run the tests in Visual Studio IDE itself, we need to create a test list as described in http://msdn.microsoft.com/en-us/library/ms182462.aspx.

Once the test list gets created, run the test by right clicking on the test list name and selecting Run. If everything goes well the Test Results will be displayed at the bottom of the screen as shown below in Figure 11.0

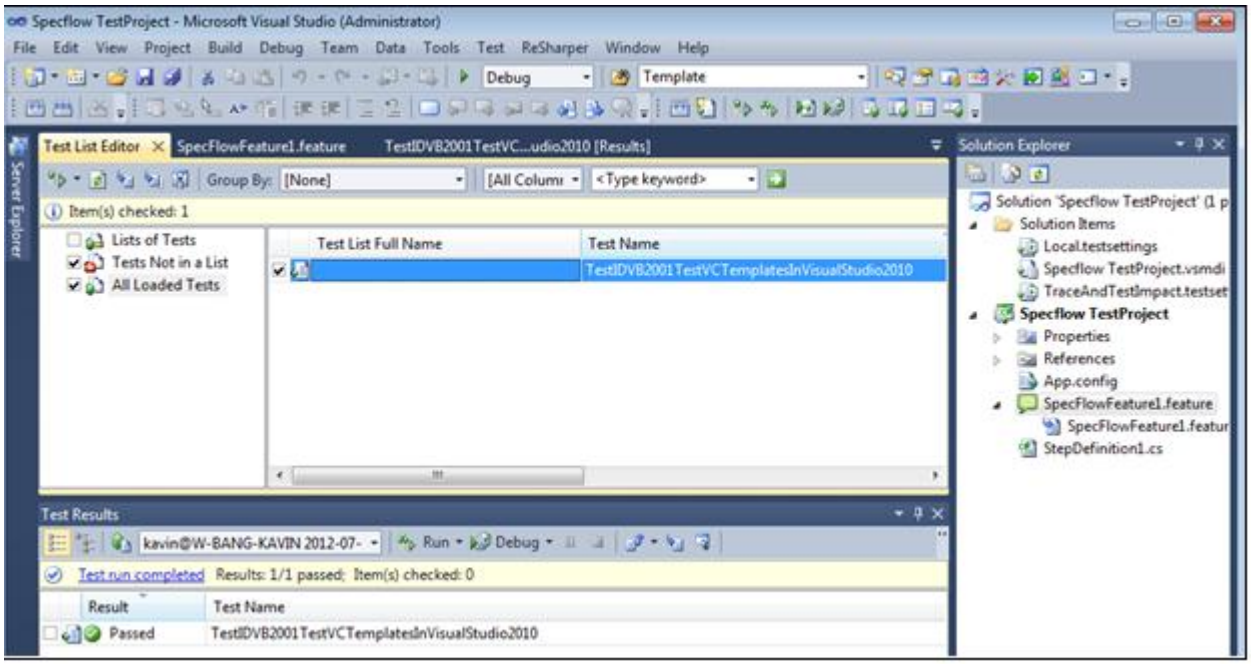


Figure 11.0 – Test Execution in Visual Studio IDE

4.2.3 Report Generation in .html format

The Test Results output from Visual Studio IDE is a .trx file. If we need an .html report from the .trx file, we need to generate it using the command mstestexecutionreport as shown below in Figure 12.0. This can obviously be automated into build steps after a set of tests have been run.

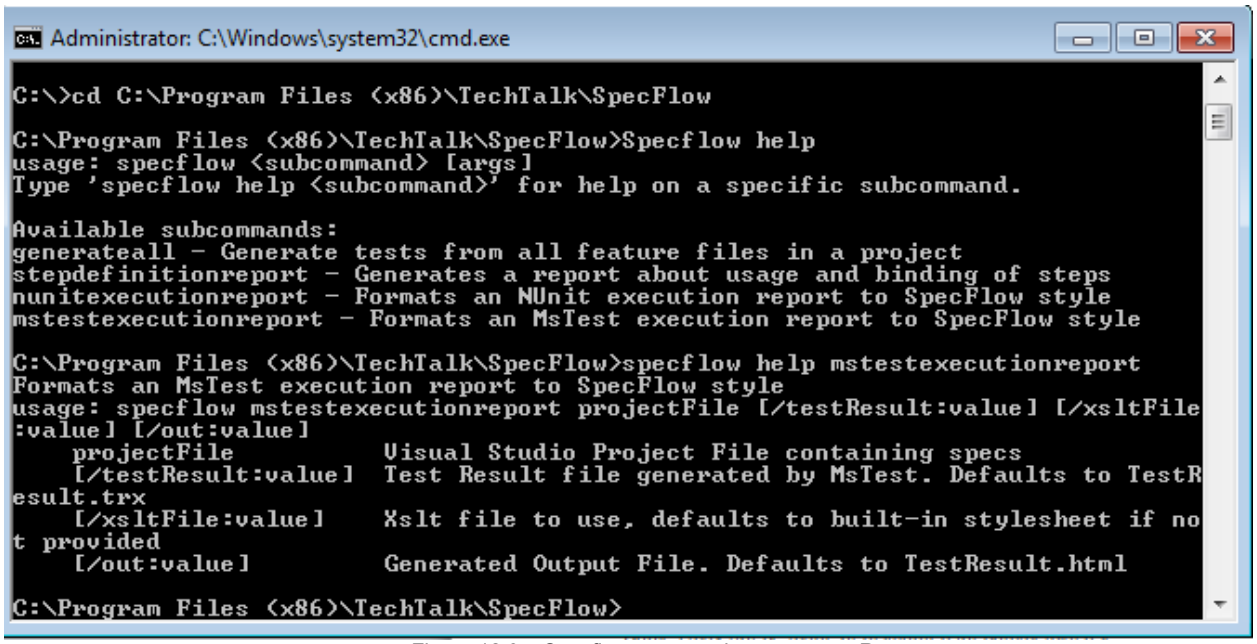


Figure 12.0 – Specflow commands to generate Report

The test report generated in .html format would look like the below screen shot as shown in Figure 13.0.

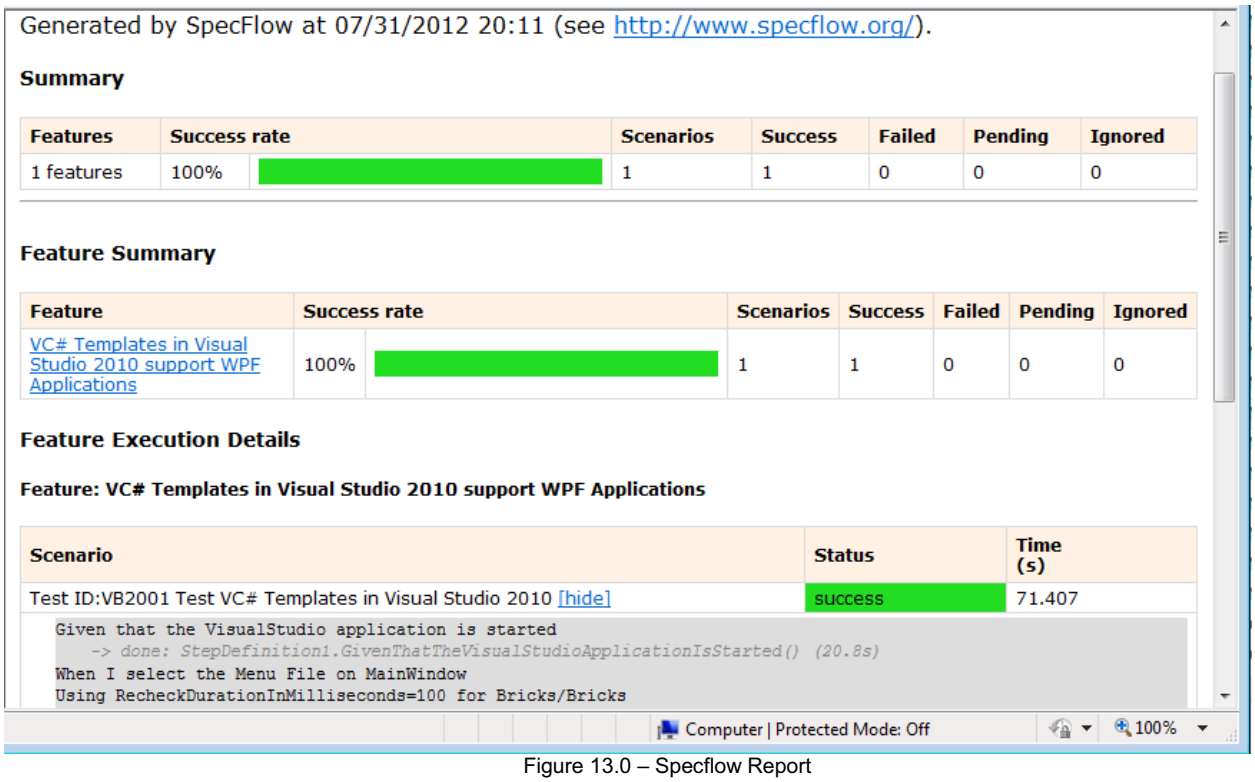


Figure 13.0 – Specflow Report

5 Project Example - Time Saved in Automation and Return on Investment (ROI)

5.1 Manual Testing

Before we suggested using SpecFlow and White, only manual testing was planned for the application under test, which in this case is a Visual Studio plugin. The same testing steps were repeated again and again. The estimate for one testing cycle was one week. There were certain areas which could not be tested manually for all the conditions which involved comparing data result values to the accuracy of nine digits! For every release, the team was spending one week's time in testing activities.

Since one week of testing was required to complete one cycle of testing, testing all the functionalities on the daily builds of the software was impossible. The team decided to do just two rounds of testing per month. Due to this, defects could not be caught on daily builds of the application under test.

5.2 Automated Testing

After exploring ways to automate the VS2010 plugin, all the functionalities of the application could be tested in a day. Testing tasks which were taking days to complete got completed in hours' time. Hence, the automation scripts were integrated with the daily builds. The result details were sent as an e-mail attachment to all the stakeholders on daily basis. The entire team was getting updated on the testing details.

5.3 Return on Investment (ROI)

The table below gives the data as to how much time and cost we save after automating the tests using SpecFlow and White.

In all companies, management needs high output with less cost for every project done. By calculating the ROI for automation and projecting the details as given in this table, we are able to demonstrate real benefits we reaped in carrying out automation using open source tools.

5.3.1 Time Saved in Automation

Functionality	Manual	Automation
VS2010 template testing and verification on application under test	360 min	25 min
Backward compatibility	240 min	25 min
Multiple template loading and verification on application under test	480 min	25 min
Data verification testing	360 min	60 min
Functionality testing	360 min	25 min
Total Time	1800 min(30hrs)	95 min

Figure 14.0 – Time saved in automation

Cost of manual testing per annum		
# test cycles per annum	**20**	
# hardware configurations on which test cycles to be executed	**5**	
Manual testing effort per cycle per hardware configuration (person-days)	**5**	
Total manual test effort (person days)	**500**	
Cost of automated testing per annum	**First year**	**Subsequent years**
Tools training / learning (person-days)	**20**	**0**
Test framework & script preparation (person days)	**30**	**0**
Test framework & script maintenance (person days)	**10**	**10**
Execution & analysis of results for 20 cycles & 5 h/w platforms	**100**	**100**
Total automated test effort (person days)	**160**	**110**
Cost savings (considering both automation & open source tool usage)		
Due to effort at $40 / hours ($) & considering 8 working hours per day	**108,800**	**124,800**
Due to open source tool usage in lieu of licensed tool ($)	**10,000**	**10,000**
Total savings ($)	**118,800**	**134,800**

Figure 15.0 – ROI calculation

5.3.2 Some Additional Benefits of Test Automation:

1. Faster execution of the tests and elimination of human errors.
2. Systematic testing process.
3. Maximized test coverage which will assure the quality of the deliverables consistently.
4. Repeatability of test execution.
5. More focus on testing new features.
6. Tests can be re-used on multiple hardware platforms and configuration.
7. Enhanced product quality.
8. Test scripts integration with daily builds as part of continuous integration activity.
9. Intangible benefits, which include developer and tester satisfaction.

6 Conclusion

Selecting the right tool for Agile Testing Methodologies is very important. Every company wants to optimize cost at the same time deliver a quality product. Hence, test engineers need to be very dynamic and explore several options to reduce test cycle time for repeated procedures. Industry standard tools are very expensive and add to cost year on year basis. To optimize cost, we need to use open source tools and show the ROI to the management in test automation.

Acknowledgements

The authors would like to thank Ian Dees who introduced us to BDD and White and to continuously provide support and encouragement to us in all the projects that we have worked. We are also thankful for his comprehensive review of this paper. We would like to thank Richard Vireday for his valuable review comments of this paper.
And to acknowledge the great work of the Open Source teams of SpecFlow and White, especially Jonas Bandi and Gasper Nagy who answered a lot of our questions, which enabled us to set up our test framework successfully.

References

http://www.specflow.org/

http://cukes.info/

http://cukes.info/http://white.codeplex.com/

http://behaviour-driven.org/

http://dannorth.net/introducing-bdd/

https://github.com/techtalk/SpecFlow

http://github.com/techtalk/SpecFlow-Examples

http://github.com/aslakhellesoy/cucumber/wiki

http://www.marcusoft.net/search/label/SpecFlow

www.cognizant.com/insightswhitepapers/open source testing tools: the paradigm shift

CODE COVERAGE ISN'T QUALITY, IT ISN'T EVEN COVERAGE

Wayne Roseberry

wayner@microsoft.com

Abstract

If anybody ever says "I know our code has high quality because we achieved 100% code coverage," smack them. There really is no relationship between code coverage and quality because code coverage does a dreadful job telling you whether or not tests have been thorough.

Good test coverage only comes from doing as many interesting things as possible that are good at exposing flaws in code. Code coverage reports, when used as a metric of quality, hide the useful tests.

This paper shows real examples of code coverage reports that showed 100% coverage that were completely, and entirely useless. Alternate tests will be presented that demonstrate how sometimes directing one's attention in the opposite direction implied by code coverage actually yields better test generation and more confidence in what the test suite addresses.

Biography

Wayne Roseberry is a Principal Design Engineer in Test at Microsoft Corporation, where he has been working since June of 1990. His software testing experience ranges from the first release of The Microsoft Network (MSN), Microsoft Commercial Internet Services, Site Server, and all versions of SharePoint. Previous to testing, Wayne also worked in Microsoft Product Support Services, assisting customers of Microsoft Office.

Previous to working for Microsoft, Wayne did contract work as a software illustrator for Shopware Educational Systems.

In his spare time, Wayne writes, illustrates and self-publishes children's literature.

Copyright Wayne Roseberry, July 29, 2012

1 INTRODUCTION

Code coverage tools are one of the more useful devices in a tester's handbag. Knowing which blocks have been covered, or not covered, by a set of tests is very informative. This has led some people to use code coverage as a measure of quality in the product and test. The practice is simple. Run some tests, look at the coverage report. Modify the tests to hit blocks not covered and repeat until the code coverage numbers are high enough percentage to feel good.

The problem with this approach is that code coverage numbers don't tell you if testing was good. In fact, the very nature of code coverage reporting is such that it hides, obscures and misleads your impressions. You are likely led to believe that since a large percentage of blocks were hit during test that you have done a pretty good job looking for bugs, or that the equivalent set of tests when run again will do a thorough job of discovering regressions. You will likely be wrong in that belief.

The fact is code coverage reports miss several classes of bugs, and obscure the need for important and sometimes large sets of test. The proper way to use code coverage reporting to improve test coverage is not as a metric of quality, but instead as an inspiration for discovering what tests ought to be introduced.

My intent is to share some stories that demonstrate these points. The stories are real; the examples are real-world. I took the liberty to modify the code examples from the original, partly because the code is proprietary and partly for readability in the document, but the story is still the same as it played out in real life. In both stories, somebody had made a decision based on blindly following the numbers from a code coverage report and did the wrong thing with regard to tests.

2 CODE COVERAGE & TEST COVERAGE

Let's talk about what we mean by code coverage and specifically test coverage. The definition is important, because it helps us clarify our goals and intent.

What is meant by code coverage?

By "code coverage", I specifically refer to tools that can show exactly which parts of the product code, expressed as source code, were hit during a test run. The most common form of code coverage is referred to as "block coverage," which is discreet pieces of source code that have a single entry and a single exit point and no branches. For example, the following code…

```
if (a == true)
{
   return 1;
}
else
{
   return 0;
}
```

has three blocks. The first block is the `if` statement, the second block is the one that returns 1 if the statement resolves to true, and the third block is the one that returns 0 if the statement resolves to false. Assuming somebody were to test this code in the case where the variable a was true, that test would achieve 66% code coverage, two out of three blocks were hit by the test.

There are other ways to describe code coverage, path coverage, traversal coverage, line coverage. For sake of this document we are going to talk about block coverage as described above. The same principles generally apply across all forms.

Classical Approach to Code Coverage

The typical way to treat code coverage is to do the following:

1. Enable a code coverage tool, sometimes by instrumenting a build, sometimes not
2. Execute a series of tests
3. Look at the Percentage of blocks covered
4. Do a white-box inspection of the code to figure out how the blocks were missed
5. Continue building/executing new tests until the percentage of blocks covered reaches a desired goal (e.g. 80%, 100%, etc.)

Examples of this approach can be found in previous PNSQC sessions (Karwa & Panda). Teams have had varying success and value out of achieving different goals, with interesting results on diminishing returns going after the last few percentage points (Manu, Najpal, Amalo & Tan).

What do we mean by test coverage?

For sake of this document, I will use a very simplified definition of test coverage. Test coverage is a list of things that have been done to discover flaws in a product. When we talk about test coverage as a percentage, we are talking about the number of things that have been done divided by the number of things that ought to be done to discover flaws in a product. In this definition, we use the phrase “ought to be done” to acknowledge that the number of things which could be done is almost always impossible or impractical and that to accommodate we have already applied our best judgment to pick the ones which we believe increase the odds of finding the flaws we desire most to remove from the product.

Tests are typically generated by enumerating the product behaviors and exercising them under different conditions, with different inputs and in different sequences.

This simple definition makes it easy to draw a comparison between code coverage and test coverage. Code coverage describes the inner structures which control the product behaviors. It is a way of saying “here are all the blocks of code in our system,” which ought to generate the question “what are the different tests we should make based on this list of blocks?” As I will demonstrate later, sometimes we answer that question the wrong way. All too often, the answer is “I need to do something to hit that block of code,” when the real answer should be “What tests will find problems because of that block of code?”

How does “coverage” relate to “quality”?

Bypassing the debate on what we mean by quality, let’s just say that quality is “the degree to which the intended customer thinks the product is good”. I settle on that definition because it acknowledges the subjective nature of quality. I also chose it because it demonstrates that all our engineering metrics cannot truly identify quality. All they can do is attempt a prediction of the customer’s opinion of the quality of the product. Better metrics help us make better predictions, and poor metrics, or poor use of metrics lead us to make bad predictions. Better coverage metrics usually the results of a rich set of test activities that are designed to effectively and efficiently uncover the flaws in the product that will reduce quality as perceived by the customer. When a rich set of tests are well executed and passing at a reasonably high rate, then we are more able to make an accurate prediction that the product is of a quality suitable for the customer’s needs.

This is the point where people get into trouble with code coverage metrics.

3 MISTAKES, RULES AND PRINCIPLES

Two mistakes

This document is going to focus on two main mistakes that people make when dealing with code coverage numbers. The first is assuming that code coverage is an indication of product quality. The argument is that if a large percentage of blocks are hit by a given set of tests, and assuming those tests pass, then the product must be of high quality. In fact, code coverage percentage has a very weak relationship to product quality, several examples of which will be given later.

The next mistake derives from the first, that being to chase the code coverage numbers by using test activities that drive the numbers higher. This leads people to often indiscriminately take the easiest, most efficient or most expedient route to achieving those numbers. The problem with this approach is that the activity which is the most practical way to increase the code coverage numbers is very often a poor way of discovering flaws in a product. Examples will be given later in this document.

The Rules

I like to reduce the rules regarding interpretation of code coverage results to the following two simple statements:

- Missing Blocks in Code Coverage == Badness
- Hitting Blocks in Code Coverage != Goodness

It is very important to understand the subtle implications from these rules. The only qualitative statement that code coverage reports can state about a given test suite is whether or not tests have missed a block of code. We assume that all missed blocks of code carry some degree of badness because we desire, if possible, to cover every block we can. Previous works make the case for exactly how much coverage is worth achieving, typically calling the practical limit somewhere around 80% before results diminish dramatically, but it is at least logically sound to say that a missed block is something to investigate because we generally do not want to see that.

However, a code coverage report does not indicate whether or not testing is sufficient, or even good. In fact, a portion of code could show as 100% covered even if the test that achieved that end accomplished nothing useful with regard to finding flaws in the product. It is therefore unsafe to decide that since a code coverage report indicates high percentages that the quality of coverage must likewise be good.

This means that code coverage reports should not be used to assess product quality. The only thing they can measure is a negative, whether or not there is a block of code not covered yet. This is useful, but it should not be used for special means. We will talk about those later.

Metaphors:

- Assessing quality of code from a code coverage report is like assessing quality of food in a restaurant by counting the number of employees who show up for work when scheduled
- People behavior according to measurements: My uncle Jerry used to sit on his seat belt with it buckled underneath him to avoid the warning buzzer.

4 EXAMPLE 1: CHASING THE NUMBERS

My first example came while covering part of the list editing and management features in SharePoint Foundation. A tester working for me was using code coverage reports to choose how to improve the existing set of automated tests. The report showed some blocks not covered, so based on that he had decided to target those blocks. I asked him to show me an example of what he meant. The code he showed me looked something like the following:

```
private object returnfielddata(fieldobject fld)
  {
    switch (fld.type)
    {
      Case FLD_STRING:
        return ExtractStringFromField(fld);
      Case FLD_DATE:
        return ExtractDateFromField(fld);
      Else
        return null;
    }
  }
```

The code coverage report said that this function had 75% coverage. It divides into four basic blocks, shown below:

Covered	`switch (fld.type)`
	`{`
Covered	`Case FLD_STRING:`
	`  return ExtractStringFromField(fld);`
Covered	`Case FLD_DATE:`
	`  return ExtractDateFromField(fld);`
Not Covered	`Else`
	`  return null;`
	`}`

In the code above, the variable `fld` is a class of type `fieldobject`. The `Type` property of this class is an enumerator of which there are around a dozen potential values (e.g. `FLD_INTEGER, FLD_FLOAT, FLD_CURRENCY`, etc.). The function is a private function, meaning only the application itself would be able to call the code.

I asked the tester what he intended to do. His response was the following:

1. Build his own version of the application in which he would put in a hook to write his own test code

2. Write a test method that would call the function directly, setting `fld.type` to some value other than `FLD_STRING or FLD_DATE`

He was seeking no other goal to than to ensure the code coverage report indicated 100% coverage. As far as he was concerned, the uncovered block, on its own, was interesting enough to merit the extra effort of writing special test hooks into the product just to allow test code to specifically call the function.

Let's look at that line of reasoning. Why is it so interesting to cover that block of code specifically? It is clear and obvious exactly what is going to happen if `fld.type` equals some value other than `FLD_STRING` or `FLD_DATE`. The function will return null. There is no other possibility. It is not like the C compiler is going to suddenly behave differently than it has for decades and cause a switch statement to evaluate differently. Testing the function in this way has no value or purpose. Yet, by assuming that the

goal of producing a code coverage report was to drive the coverage numbers as high as possible, that is precisely the kind of test this engineer was proposing. The tester was allowing the metrics to mislead him.

Something Far More Important

When I saw this uncovered block of code, I had a completely different set of concerns:

1. *Any value other than* `FLD_STRING` *and* `FLD_DATE` *were treated as an equivalence class by the code.*

   One of the biggest sources of bugs in product code, and something completely missed by code coverage reports is missing logic. What if one of the other possible values for `fld.type` needed to be addressed in the switch statement?

   Making it even more complicated, more than one test would hit exactly the same code block. Assume, for a moment that `FLD_INTEGER` should return null, `but FLD_COUNTRYCODE` should not. If during test, only the value `FLD_INTEGER` were used, the code coverage report would show the block had been hit, yet we would be no closer to discovering that the code does not properly handle `FLD_COUNTRYCODE` because from a pure code block perspective the two cases are equivalence classes, when in fact they should not be.

2. *Is* `null` *an appropriate return value for the other ranges on* `fld.type`*?*

   Again, code coverage will not tell you if the behavior in the code is correct, only if it has been executed or not. Perhaps `null` should be replaced with some other value.

3. *Our tests had never tried any values beyond* `FLD_STRING` *and* `FLD_DATE`

   As stated above, the enumerator data type for `fld.type` has about a dozen or so possible values. If the code coverage report indicated the final `return null;` block had never been hit, then that means the tests had not included any of those other values for `fld.type`. This says not so much about the function coverage itself as it says about coverage of the rest of the product code. The actual code coverage report is a hint that something is wrong, but the missed block is not the appropriate target of our attentions. The missing block is telling us there is something wrong somewhere else.

4. *The consuming code of the function "returnfielddata" had never processed a return value of null when under test*

   Imagine that the code calling this function looked like the following:

   ```
   fielddata fd = (fielddata) returnfielddata(field);
   addFieldToDocument (fd.value);
   ```

   We don't need to know what `fielddata` is, or what `addFieldToDocument` does to recognize that if returnfielddata returns a null object into `fd` that the next statement is going to fault when trying to access the `Value` property on `fd`.

5. *The switch statement where uncovered block was an else clause on an enumeration, and it returned a NULL*

   PRINCIPLES:

   - The uncovered block isn't interesting in itself

- Taking gyrations to run the block in isolation (e.g. special build to make the internal API callable and code block easily hittable)
- REAL PROBLEM: The test code isn't trying all value possibilities in the enumeration
- REAL PROBLEM: Hitting only one value would give illusion of 100% coverage
- REAL PROBLEM: The calling code has never been tested with the value returned in the else block – e.g. what if return was `null` and the caller immediately tries to use `result.member`

5 EXAMPLE 2: GOOD ISN'T GOOD

I experienced a similar issue several months later, around the same feature set. I assigned one of my testers a piece of code that was used for building queries to retrieve items from the database. The previous owner had written automation and achieved 100% coverage of a particular class. I had been told by the previous feature owner the coverage was in pretty good shape because of the numbers on the report. I asked the tester to evaluate the test automation to ensure it was good. Several days later, she came back to tell me she had found some substantial problems, in spite of the 100% coverage stated in the report.

The feature in question was for a class called SPQuery. It is mostly a container that is used to store an XML description of which items to retrieve. There are additional properties on the class that indicate how the retrieval should behave. On its own, the SPQuery class does not do anything. After the SPQuery object is constructed and its arguments set, it is passed to a method called "GetItems" that converts the SPQuery object into an SQL database query and performs the actual fetch. A typical pattern for using this class would look as follows:

```
// instantiate the object, set its properties
SPQuery qry = new SPQuery();
qry.Query = queryXMLString; // assume this was set prior…
qry.DatesInUTC = true; // we want date fields to be in UTC format
qry.AutoHyperlink = true; // we want HTML format anchor tags

// fetch the items from our list by passing in our query object
SPListItemCollection items = splist.GetItems(qry);

// once here, the code will iterate through results in items object
…
```

The tester assigned to the project pointed out to me that the high coverage numbers had been achieved simply by instantiating the class and checking that the properties were being set properly. Here is an example of the sort of test code she found:

```
// instantiate the object, set its properties
SPQuery qry = new SPQuery();
qry.Query = queryXMLString // assume this was set prior…;
if (qry.Query == queryXMLString)
{
   Log.Pass("Query matched expected value");
}
else
{
   Log.Fail("Query did not match expected value");
}
```

Likewise:

```
// instantiate the object, set its properties
SPQuery qry = new SPQuery();
qry.DatesInUTC = true; // assume this was set prior…
if (qry.DatesInUTC)
{
   Log.Pass("DatesInUTC set as expected");
}
else
{
   Log.Fail("DatesInUTC not set as expected");
}
```

By doing the above, the previous tester had very quickly and very efficiently achieved 100% code coverage of the entire SPQuery class. They had also accomplished very little. For example, in the above property `DatesInUTC`, the product code looked something like this:

```
public bool DatesInUTC()
{
   get {return m_DatesInUTC;}
   set {m_DatesInUTC = value;}
}
```

All that happens when the property is either assigned or retrieved is it is put into, or read from, an in memory member property of the class. This is an exceptionally uninteresting thing to test. There are no transformations, there is no parsing, the data does not get passed to another process or method during either assignment or retrieval, and the behavior does not change based on any condition or state.

The tests which had achieved 100% code coverage on this class were almost entirely useless. They were doing no physical harm, although they were doing a bit of psychological harm by creating the illusion that the class had been well tested and covered.

Fortunately, the tester I had asked to analyze the code realized this, and came up with the following conclusions:

1. **Using Wrong Test Pattern**

   The automation was not exercising the expected pattern for this class. The SPQuery class is a primary input to the `GetItems` method, and therefore it was really only interesting to test it by setting the class properties and subsequently calling `GetItems()`.

2. **Missing Properties That Drive Behavior**

   The automation was not testing how different properties on the SPQuery object affected behavior when subsequently calling `GetItems`. The `DatesInUTC` test stated above is a good example. The property affects whether the data returned from `GetItems` will come back in the format of a URL (e.g. http://www.contoso.com) or anchor tag (e.g. "http://www.contoso.com"). In the very least, there should have been two tests for `DatesInUTC`, one where it was set true, and one where it was set false. Further, assume that the developer had forgotten to format results as an anchor tag and always returned hyperlinks in their raw format. A code coverage report of such a mistake would likely show the block completely covered, as such bugs are missing code blocks, something code coverage is incapable of detecting.

3. **Missing Large, Complex Existing Data Domain**

The SPQuery and `GetItems` methods return items from lists of data with customized column data types (string, date, integer, people, lookups on other columns, hyperlinks, etc.). Furthermore, those lists can have different settings that affect retrieval behavior (do they contain subfolders, do they items have multiple versions, security settings on this list, etc.). These different data domains represent a large number of conditions and states that affect the behavior of the `GetItems` method, and integrate very tightly with the state of the SPQuery object when `GetItems` is called. The test automation was missing tests that populated lists with different values, field types and settings.

4. **Missing Large, Complex Format and Behavior Domain**

   The SPQuery class takes as its primary property a string that describes in XML the items that ought to be retrieved from the list. This XML query is transformed into an SQL database query. The XML syntax for the query is moderately complex; the generated SQL query is extremely complex. The SQL query generation and execution is one of the most complicated features in the product overall, making the development team very nervous about fixing any bugs. Regressions in this feature are notoriously difficult to find, understand and fix. Yet, despite the complexity of this problem, the test automation was not trying variations on the XML query format.

   One of the riskiest and most difficult portions of the code was being virtually unexercised because a code coverage report had said a feature was 100% covered. One important point to note is that a large part of the code logic happens inside the generated SQL, something the code coverage tools we were using was unable to inspect. With regard to different XML string, as far as the code coverage was concerned they were all equivalent data, because the get and set methods on the Query property did so little.

Working together, the tester and I drew the following conclusions for what to do:

1. Enhance the test tools to more easily generate lists with the different range of data types
2. Build data generation tools that populated lists with interesting test data ranges
3. Build query XML generating tools that could build thousands of different queries in different structures
4. Build a simple query testing validation mechanism that combined the three components above to easily cover lots of different lists, with lots of different data and settings with lots of different types of queries

6 WHAT SHOULD WE DO?

So, what is our call action? If code coverage reports are a poor assessment of quality, and if using them to chase the numbers and achieve high coverage percentages the most efficient, easy way possible leads to insufficient and sometimes misleading results, then what should we do instead?

Fortunately there is an answer, and perhaps also unfortunately it isn't a simple, easy-as-pie thing to do. Like most aspects of software test engineering, the right thing to do is hard and requires skill and experience. But then again, that is largely what makes the job as rewarding as it is. Here are the recommendations that I would extend based on the experiences I have described so far:

- **Use code coverage reports like a probe**

  Code coverage reports are an essential tool. They help you see something that is otherwise unavailable to you. If feasible, run them as often as you can with the narrowest granularity possible. In a lot of ways, code coverage reports are like the gas gauge on a car, or the

thermometer outside. The gas gauge will not tell you if the trip you are driving on is successful no more than the thermometer will tell you if you are having a good day or a bad day. But both are critical tools for driving somewhere and having a picnic (or a snowball fight) respectively. Check the code coverage reports periodically and use them to help guide adjustments in your test approach.

To complement this, if you are using code coverage reports as a means of measuring product quality, replace it with something else, such as bug open and fix rates, customer feedback and incident reports, reliability reports and test case execution pass fail rates. Whatever it is you use, you want it to have more success predicting impact on the customer.

- **Consider every missed block bad**

How bad a missed block might be is up to you, but your default disposition ought to be that a missed block of code is innately undesirable. Maybe the best way to address this would be to state the report the opposite way we are used to. Most of the time, we state code coverage in terms of percent covered (e.g. “We achieved 75% coverage!”). Perhaps it is better to state in terms of percent uncovered (e.g. “We still have 45% of our code blocks uncovered in testing”).

Do not do the opposite and consider every covered block good. To that, see the next point.

- **Ask “What does this mean?” rather than “How do I hit this block?”**

Code coverage reports should more than anything else inspire imagination and new ideas. They help you understand things that are not happening, and it is in those gaps where we sometimes learn the most important things.

Ask two questions: 1> “What are the tests doing that hit the covered blocks?”, and 2> “What do the uncovered blocks imply regarding missing tests?” For each of these, force yourself to ignore the opportunity to “just make sure the block gets hit” and instead try to think of the following:

Covered blocks:

- Create Strong tests that cover many domains and protect the code from regression
- Identify and remove weak tests that achieve superficial depth and are poor at discovering bugs
- Target tests that may be selected when specific portions of the code change
- Develop opportunities to borrow between tests for things like shared data, configuration, and system state, sequences of operations, etc.

Uncovered blocks:

- Identify weak investments in larger test domains, such as heavily data dependent, system state dependent
- Call attention to weakly covered features that directly or indirectly integrate with the missing block and do not handle variations in data, return values, error states and faults or other implications of the missed block
- Correct or replace test patterns that do not exercise the product appropriately to force integration of components or usage the way intended
- Seek opportunities for tests in usage scenarios

All of the above may sound easier said than done, because in fact they are. This is an activity that improves with experience. It is the sort of thing where more senior engineers and leads ought to mentor others in how to come up with better ways to use the code coverage information.

7 CONCLUSION

Quality metrics are an artifact of our constant search for ways to manage our schedules and predict how well a product will do in the hands of a customer. Sometimes we come across a useful tool that makes us do our job better and surrender to the temptation to turn that tool into a quality metric. Sometimes that works really well, guiding us to more successful projects with better product, and sometimes it introduces problems that hinder us more than it helps. Code coverage reports are that sort of useful tool. They are an essential part of every test engineer's toolkit, but as demonstrated in the stories shared in this document, when used as a metric of product quality the unintended side effects are shortcuts and misleading coverage. For sake of better software engineering, code coverage reports are best treated as a tool carried in the engineer tool belt and not plastered on the wall of the executive boardroom.

References

Manu, C., Najpal, P., Amalo, D. & Tan, R.P. "Code Coverage Case Study: Covering the Last 9%". Paper presented at the Pacific Northwest Software Quality Conference, Portland, Oregon, October 2010.

Karwa, S. & Panda, S. "Streamlining test Automation using White-box Driven Approach". Paper presented at the Pacific Northwest Software Quality Conference, Portland, Oregon, October 2010.

Microsoft. "SPQuery Class" Published 2010 http://msdn.microsoft.com/en-us/library/microsoft.sharepoint.spquery.aspx

Introducing Static Analysis in a Mature Codebase

Author
John Ruberto
john@ruberto.com

Abstract

This paper tells the story of how our organization introduced static analysis into its software development process. The static analysis process automatically finds bugs, assigns them to the engineer that most likely introduced the bug, and then automatically verifies the fix.

This project began two years ago, when we started recording the root cause of bugs that were fixed. Once we had several hundred bugs analyzed, we found that the largest number of bugs was caused by coding errors. The good news, there are many best practices in industry to improve code quality. One survey of software quality practices (Jones 2011) shows that three practices, taken together, are capable of removing 97% of the defects. Those three practices are code review, static analysis, and unit testing.

Our organization already had healthy processes for code review and unit testing, but we had spotty history with static analysis. So, we took a fresh look at static analysis. When engineering the process for static analysis, we took care to avoid some of the issues of the past; namely false positives, excessive effort required to manage the bug list (triage false positives, manage assignments, etc.), and the demotivation of analyzing a large backlog of defects.

When we designed the new process, we decided to focus on automating the bug identification, assignment, and retest processes – eliminating a lot of the effort that got in the way of previous attempts. We also focused on keeping new code clean, which is much easier to justify than clearing old bugs from the backlog. The process also was designed to provide fast feedback, directly to the developer that created the bug.

Paying attention to new bugs has resulted in high levels of adoption by the developers, as compared to focus on backlog bugs. The engineers are fixing the new bugs that come in, plus have made a significant dent in the backlog.

Biography

John Ruberto has been developing software in a variety of roles for over 25 years. Currently, he is the Quality Leader for QuickBooks Online, a web application that helps small business owners manage their finances.

John has a B.S. in Computer and Electrical Engineering from Purdue University, an M.S. in Computer Science from Washington University, and an MBA from San Jose State University.

Copyright John Ruberto June 2012

1 Introduction

My personal experience with using static analysis for established projects has been very spotty. These projects, which have a considerable amount of pre-existing code, are successful in their respective markets. Running static analysis tools on these projects usually produces an overwhelming volume of potential defects. Since the product is successful, the static analysis results are doubted.

In one example where I introduced the used of a static analysis tool, I was the test leader for a 5 million line C++ system with a client/server architecture. The project had been in place for seven years, and was successful in the market. I attempted to introduce static analysis into our software development process, with the goal of reducing defects found in the system test phase.

The static analysis tool that we chose found over 400K errors in the first pass. This seemed like a great tool and would really help us, until we tried to get developers to look at the result. The vast majority of the bugs did not affect functionality, but were related to odd code structure. These bugs could have caused future maintenance issues, but were not clearly bugs that needed to be addressed today. A majority of the errors that could affect functionality were false positives. A false positive bug is where the tool indicates a bug is present, but actually is the intended implementation and is not a bug after all.

Also, most of the errors found were in the legacy code, which was running in production and was successful in the market. These factors made it difficult to engage engineers to invest time to examine the bugs, much less fix them.

So, when introducing static analysis in my current company and project, I was wary about introducing this process, only to have the results ignored. This paper describes the journey that our team took to successfully introduce static analysis into an ongoing successful project.

2 Why Static Analysis?

We track the root cause for bugs found in system test, and through Pareto analysis (Bose 2010) noted that “coding errors” were the most frequent type of errors. Figure 1 shows a pie chart depicting the various causes of bugs found in system test. Coding errors were the top contributors.

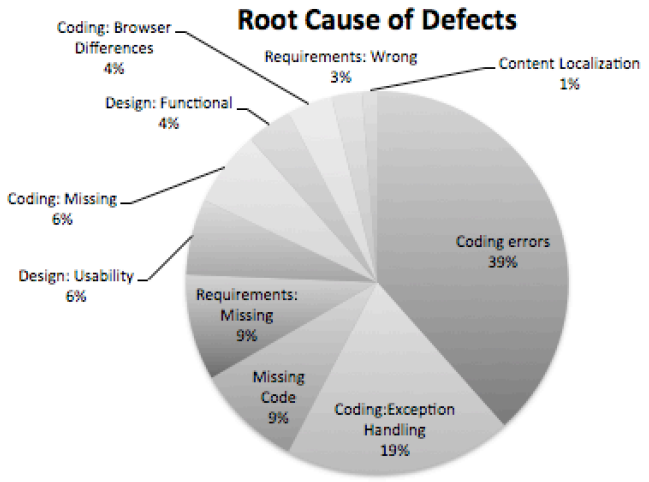


Figure 1: Pie chart of the causes of bugs in our application. This chart shows that coding errors are the most frequent cause of bugs.

The good news is that there are many best practices in industry to improve code quality. One survey of software quality practices (Jones 2011) shows that three practices, taken together, are capable of removing 97% of the defects. Those three practices are code review, static analysis, and unit testing.

Our organization already had healthy processes for code review and unit testing, but we had spotty history with static analysis. So, we took a fresh look at static analysis. When engineering the process for static analysis, we took care to avoid some of the issues of the past namely false positives, excessive effort required to manage the bug list (triage false positives, manage assignments, etc.), and the demotivation of naming a large backlog of defects.

2.1 What is Static Analysis?

According to (Black, 2011), static analysis is "*Analysis of software artifacts, e.g., requirements or code, carried out without execution of these software development artifacts. Static analysis is usually carried out by means of a supporting tool.*"

Figure 2 shows an example of a defect found through static analysis. The tool used for this example was Coverity® Static Analysis for Java. In this example, the tool was able to detect that an object was being dereferenced before it was being checked to see if it had a value equal to null. If the object was null coming in, this code would have generated a null pointer exception, and the effort to create a check for the null condition would not be effective.

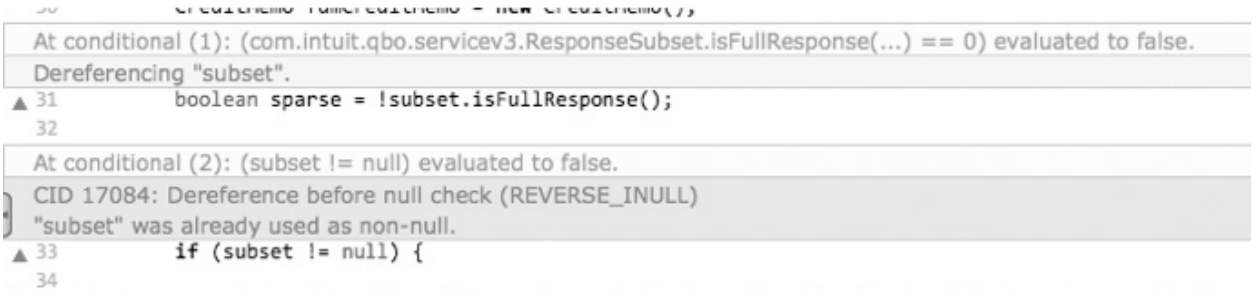


Figure 2: An example bug found by static analysis. In this bug, the object "subset" is dereferenced in line 31, and checked to see if its value is null in line 33. If the object was indeed null, this code would have thrown a "null pointer exception" before the null check happened.

This is a good example of the power of static analysis; the tool can find common coding errors like this automatically. Finding this type of error through test may be difficult if it is difficult to create the conditions where the 'subset' is null.

3 Process Design

The team formed to design and implement this process was comprised of quality engineers and developers. Before jumping into tool selection, the team designed the process. The process design illuminated requirements for the tool selection.

Using Design For Delight (Ruberto 2011), the team identified important factors for the static analysis process. The Design For Delight method involves identifying the key pain points for customers, and designing a process to solve those pain points.

In this context, developers in our department were the “customers”. The developers were very interested in delivering bug free code, and less concerned with fixing old code that was already in production. So, we designed the static code analysis process around identifying newly introduced bugs.

Plus, the new bugs were shown to be quicker and easier to fix, since the developer that introduced the bug was still assigned to the project, and the code was still fresh in their minds.

3.1 Initial Static Analysis Process

We initially designed the process with the following factors in mind:

- Find new defects as close to creation/injection as possible
- Alert the developer about the defects, and allow developers to fix defects before checking in code, or before code review.
- Track defect status and provide status reports
- Efficient management of the defect management process. Automatically assign defects to developers, and automatically close defects once they are fixed.

Figure 3 depicts the process that was designed. The static analysis happens before check-in, and ideally all defects are fixed before the code is committed to source control.

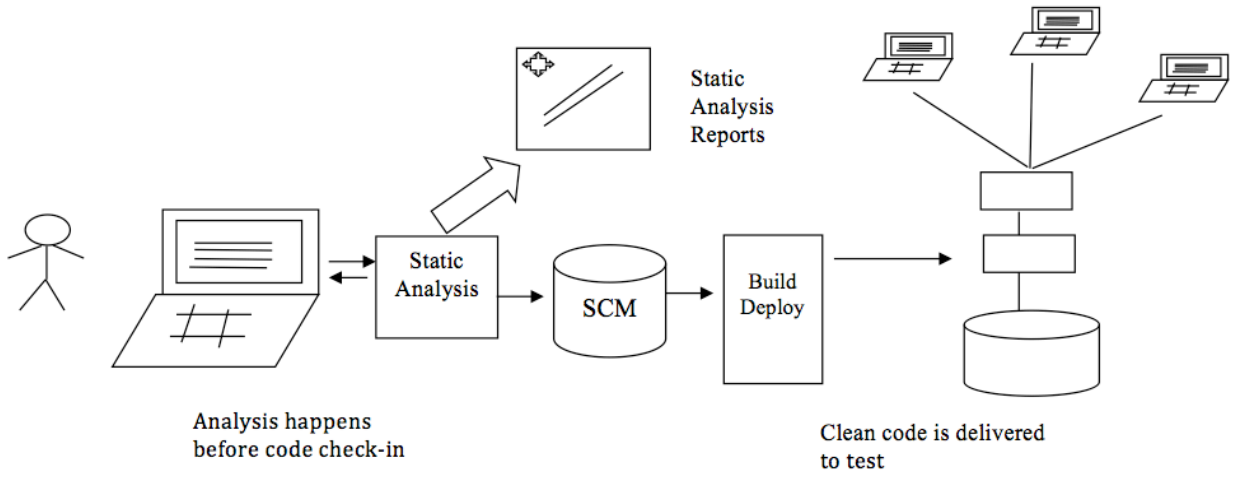


Figure 3: Incorporating Static Analysis in our build process

3.2 Process Design, Iteration 2

After surveying and evaluating the tools available, we did not find one that enabled the developer sandbox scans. The best tools actually scan the entire code base, which is only feasible in context of a full build. So, we iterated on the process, which is shown in figure 4.

This process performed the static analysis after check-in and in parallel to the build/deploy process. This process has the disadvantage of:

- Defects could affect the test, as they are detected in parallel with build/deploy
- Notification and follow-up with developers was more important, to make sure they follow up and fix the defects before moving onto other tasks.

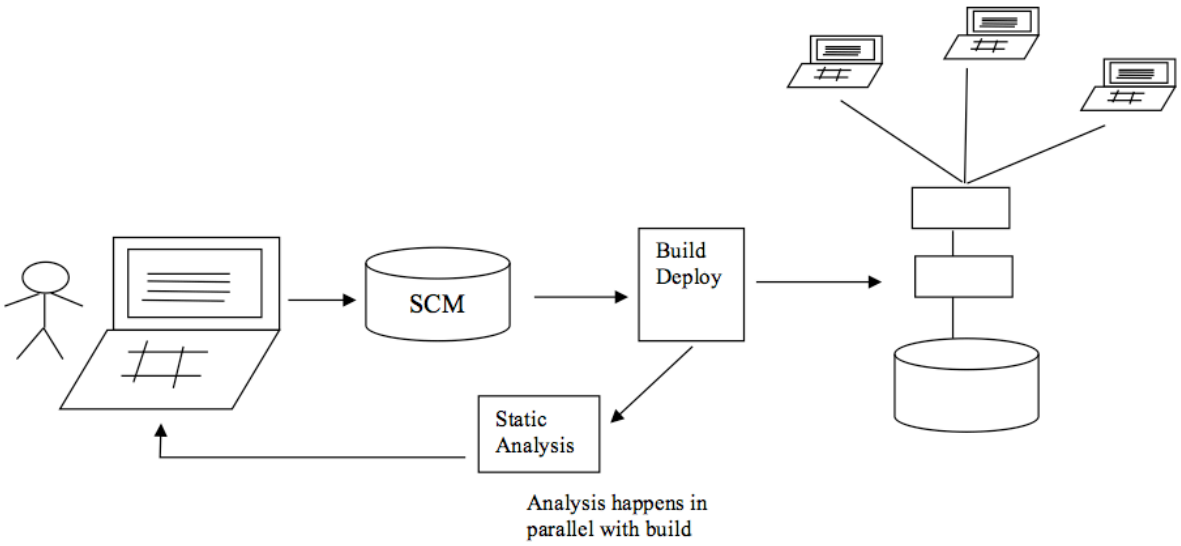


Figure 4: Modified process after learning about tool capabilities.

4 Implementation

4.1 Tool Selection

Designing the process first helped identify the requirements for tool selection. The key requirements for our search were:

- Effectiveness with our development language (Java)
- False positive rate with our code base
- Compatibility with our development environment (IDE, build system)
- Ability to uniquely identify defects
- Either a built-in defect management system, or the ability to integrate with our system through an API

Our favorite web search engine helped identify a list of potential tools. We chose a selection of industry leading tools and open source tools to actually try. It's important to try before you buy, to avoid "shelf-ware". We found a few tools that were just not compatible with our code base, even with a lot of effort and help from the vendor support team. There were also tools that worked, but produced results that we felt were not worth the effort.

Once we had scans from several tools, it was time to engage with developers. We took random samples of defects from each tool and asked developers to analyze the results. We graded several factors:

- Efficacy of the checkers (how powerful the tool is for finding interesting defects)
- False positive rate (tool indicates a defect, where there is no defect)
- False negative rate (tool doesn't find a defect, which indeed exists)

After the evaluation, we found two tools that worked well with our code base, and these tools worked well together. They had very complementary checkers (a defect missed by one tool was found by the other, and vice-versa). Table 1 shows the results of the evaluation of these two tools.

Resource Leaks	Findbugs	Coverity
OS com.intuit.qbo.bl.payrollform.EditPayrollForm.getBytesFromFile(File) may fail to close stream [line 209]	✔	✔
OS com.intuit.qbo.controller.util.AuthHeaderHelper.getResponseBodyAsString(HttpMethod) may fail to close stream [line 413]	FP	✔
com.intuit.qbo.monitor.LoadBalance.runCommand(String[]) may fail to close stream [line 214]	FP	✔
com.intuit.qbo.service.provisioning.partnerconfig.PartnerConfig.initialize() may fail to close stream [line 354]	✔	✔
OS com.intuit.qbo.ui.common.UICommonFunctions.includeUIFiles(String) may fail to close stream [line 1604]	✔	✔
OS com.intuit.qbo.ui.help.index.SiteIndex.run() may fail to close stream [line 252]	✔	FN
com.intuit.salsa.ui.login.CachedAuthCompanyInfo.loadFromFile(String) returns from routine while resource still open [line 69]	FN	✔
com.intuit.qbo.xml.xmls.XMLVendor.main(String[] args) returns from routine while resource still open [line 1104]	FN	✔
com.intuit.qbo.util.ResourceAccessGateKeeperDisableTest.touchResourceFile() returns from routine while resource still open [line 416]	FN	✔
com.intuit.qbo.util.ResourceAccessGateKeeperDisableTest.touchResourceFile() returns from routine while resource still open [line 605]	✔	✔

Possible Null Pointer Dereferences	Findbugs	Coverity
Possible null pointer dereference of defaultServiceItem in com.intuit.qbo.bl.comgmt.CreateCompany.addDefaultTimeItemToPrefs()[line 580]	FP	✔
Possible null pointer dereference of zipCode in com.intuit.qbo.bl.comgmt.QBOCompanyManager.postCreationSetup(QBOCompanyManager$CreateCompanyDataIf, boolean) [line 1953]	✔	FN
Null pointer dereference of aDB in com.intuit.qbo.bl.recur.EditRecurInfo.main(String[]) [line 917]	✔	✔
Possible null pointer dereference of response in com.intuit.qbo.bl.txns.TxnSmsNotification.invokeRequest(HttpClient, String, String) [line 244]	✔	✔
Possible null pointer dereference of checkDate in com.intuit.qbo.ui.transactions.payroll.payemployees.PayEmployeesConstants.JAVASCRIPT_SEND_PAYCHECKS(Integer, Integer, QBDate, QBDate, QBDate)	✔	FN
com.intuit.qbo.service.util.mapping.SalesReceiptMapper. mapSalesReceiptCdmType2QboType(SalesReceipt) possible null pointer dereference of detail [line 124]	FN	✔
com.intuit.qbo.ui.interviews.mini.MiniInterviewHandler.handleRequest() possible null pointer dereference of first_mip [line 135]	FN	✔
com.intuit.qbo.ui.mobile.IPhoneHandler.setLoginHelpMessage() possible null pointer dereference of locale [line 3616]	FN	✔
com.intuit.qbobilling.util.RetailRenewalUtil.send(int, boolean) dereference before null check [line 270]	FN	✔
com.intuit.qbo.ui.transactions.purch.creditcard.core.CreditCardhandler.handleRequest() dereference before null check [line 144]	FN	✔

Table 1: Matrix of defect types and score from two static analysis tools. FP = False Positive, FN = False Negative, check(✔)=valid defect (Batas 2011)

The following table lists the top fifteen most common errors detected by static analysis tools we applied to our codebase. With each error is a reference to detailed information about that error.

Defect Type	Description
NULL_RETURNS	http://cwe.mitre.org/data/definitions/476.html
FORWARD_NULL	http://cwe.mitre.org/data/definitions/476.html
RESOURCE_LEAK	http://cwe.mitre.org/data/definitions/404.html
REVERSE_INULL	http://cwe.mitre.org/data/definitions/476.html
SE_BAD_FIELD	http://findbugs.sourceforge.net/bugDescriptions.html#SE_BAD_FIELD
NP_BOOLEAN_RETURN_NULL	http://findbugs.sourceforge.net/bugDescriptions.html#NP_BOOLEAN_RETURN_NULL
SE_COMPARATOR_SHOULD_BE_SERIALIZABLE	http://findbugs.sourceforge.net/bugDescriptions.html#SE_COMPARATOR_SHOULD_BE_SERIALIZABLE
EC_NULL_ARG	http://findbugs.sourceforge.net/bugDescriptions.html#EC_NULL_ARG
LI_LAZY_INIT_UPDATE_STATIC	http://findbugs.sourceforge.net/bugDescriptions.html#LI_LAZY_INIT_UPDATE_STATIC
MF_CLASS_MASKS_FIELD	http://findbugs.sourceforge.net/bugDescriptions.html#MF_CLASS_MASKS_FIELD
GUARDED_BY_VIOLATION	http://cwe.mitre.org/data/definitions/366.html
EC_UNRELATED_TYPES	http://findbugs.sourceforge.net/bugDescriptions.html#EC_UNRELATED_TYPES
XSS_REQUEST_PARAMETER_TO_SERVLET_WRITER	http://findbugs.sourceforge.net/bugDescriptions.html#XSS_REQUEST_PARAMETER_TO_SERVLET_WRITER
GC_UNRELATED_TYPES	http://findbugs.sourceforge.net/bugDescriptions.html#GC_UNRELATED_TYPES

Table 2: The top fifteen most common errors identified by the static analysis tools we applied to our codebase

4.1.1 False Positive Rate

A high rate of false positives will destroy any hope for adoption of static analysis, and for good cause. If the tool is giving more false results than true, the value is diminished. Our evaluation paid special attention to the false positive rate.

We took several random samples of defects identified and had developers evaluate each one to determine false positive rate. Any tool with lower than 50% rate was rejected. The consensus among the

development community was that a 20% false positive rate was tolerable, especially if the tool identified many serious bugs that may have otherwise gone undetected.

4.2 Automation

Most of the static analysis packages allow integration with the build system to automatically. We took advantage of this feature as well. We also wanted to automate the defect management process.

One sore point for previous attempts at using static analysis is the triage and bug assignment process. Finding someone to review the results, determine if the defects are real, find the right developer to fix the bug, and verify that the bug is actually fixed is a labor intensive task. Performing this role is usually thankless, and often doesn't get done.

We automated the bug management:

1. When a new defect is found (not already in the list), we have a script that does a lookup for the last person to check in the file that contains the bug.
2. The script then assigns the bug to that developer in the defect management system.
3. The developer is notified, via email, that they have a new bug assigned. The email contains a description of the bug, and a link into the defect management system.
4. If a defect no longer appears on the list, it is marked as fixed automatically.

With this process, the bug assignment and closure when fixed is completely automated. Being an automated process, there have been some glitches. Most notably, the script that does the bug assignment determines the engineer by the last person to check in a file. Sometimes, two or more engineers touch a file in a day, and this has introduced some errors in the assignment. These cases happen in a small percentage of the bugs found, and the developers simply reassign the bug to the right person. Overall, the cost saved by totally automating the bug management is much higher than the cost generated by wrong assignments.

4.3 Launch and Learn

The value for these tools is to actually fix the bugs. This is where the wider developer and QA community comes into play. It is important that the process is in place and for the tool to perform its job, but more important for the users to actually use it, and get value from, this process and tool. Launching the process entails more than just educating people on what to do and how to do it, but also why we are adding static analysis.

We launched this process with a presentation to the developers. In the presentation, we covered the following topics:

- Why we should introduce static analysis (Root cause analysis data from section 2, etc.)
- Results of the tool selection process
- Examples of the errors found by static analysis
- Testaments from early adopters (developers that helped with the selection process)
- Description of the process and what is expected of each developer
- Training on how to use the tools
- Feedback process for users to identify any issues with the tool or process

We then activated the process by enabling the build time scans and email notification.

4.4 Reports

Communication and follow up is important to make sure the new process sticks. We send out several reports regularly to keep the team informed about the status static analysis, and the value it provides.

Bugs are found on most days. When a bug is found, the developer assigned to fix that bug is automatically notified with the list of bugs assigned to him or her. These reports go out daily, and are private emails directly to the developer.

The quality manager sends out a weekly status report that shows the open bugs, which developers are assigned, and the severity level of the open bugs. These reports focus on the current project. Table 3 shows an example of the weekly report.

Owner	Major	Moderate	Minor	*Total*
Amit	1			*1*
Andy		1	1	*2*
David		2	1	*3*
Frank	2		1	*3*
Mary			2	*2*
Praveen	1	1		*2*
Sandeep	1	2	1	*4*
Sunil			1	*1*
Victor	1	1		*2*
Total	***6***	***7***	***7***	***20***

Table 3: Typical Weekly status report showing open bugs by developer for the current project.

The quality manager also sends out a monthly report that shows:

- Overall backlog status trend
- Status on the previous release (How many bugs were found and how many fixed)
- Highlights the top developers for fixing backlog bugs

5 Results

Figure 5 shows the defect fix rate overlaid on the incoming rate (starting from when we deployed the tool). This graph shows that the fixes are keeping up with the arrival rate, and that we've made a dent in the backlog.

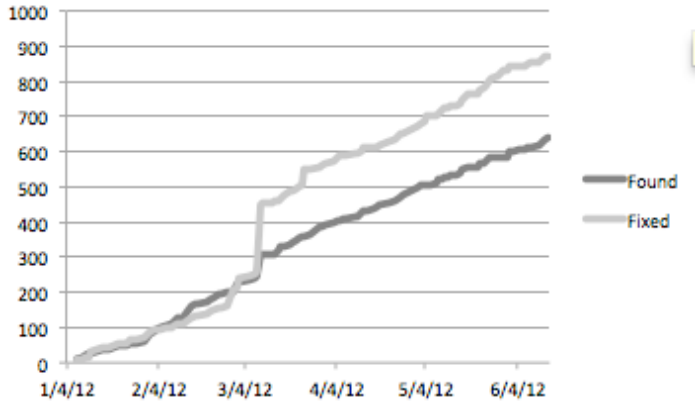


Figure 5: The defect arrival and fix rate since static analysis became active.

In the last full release, the team fixed 97% of the issues found, with 100% of the major severity defects fixed.

Since this process has been deployed, we've also reduced the number of errors and exceptions that happen in production by 20%.

Qualitatively speaking, the developers appreciate how this process has been designed. They see static analysis as a tool to help them code better.

6 Conclusions

Incorporating static analysis for this project has been a success. I believe this was successful for the following reasons:

- The process was designed to handle newly introduced bugs, rather than tackle a mountain of backlog bugs. The new bugs are the most valuable to fix since they have a high uncertainty factor and the code is still fresh.
    - New run-time errors occurring in production have declined by 20%
    - Project teams fix 97% of the identified defects prior to release
    - The bug fix rate exceeded the bug arrival rate. Over a 6-month period, approximately 600 new bugs were identified and fixed, along with 250 legacy bugs.
- Care went into the tool selection to find the best set of tools for our particular code base
    - Two tools were selected because they identified a complimentary set of defects. The chosen tools were: FindBugs, an open source utility; and the commercial Coverity® Static Analysis for Java.
    - These tools have demonstrated a false positive rate of less than 20%.
    - The low false positive rate coupled with the lightweight defect management process has improved the adoption by developers.
    - Influential developers participated in the tool selection process, which also helped to drive adoption.
- Defect management (assigning bugs, verifying closure) was completely automated, which speeds up the feedback cycle (defect injection until defect assignment), and lowers the operating cost of this process.
- Frequent reports showed the value of static analysis, which reinforced the new process

References

Batas, Brent, 2011, *Final Report on Internship Experience*

Black, Rex; Mitchell, Jamie; *Advanced Software Testing – Volume 3*, Rocky Nook, page 264

Bose, Tapan K., 2010, *Total Quality of Management*, Pearson Education India, page 343

Jones, Capers, 2011, *Software Quality in 2011, A survey of the State of the Art*.

Ruberto, John, 2011. *Design for Delight Applied to Software Process Improvement, Pacific Northwest Software Quality Conference Proceedings, 2011*, page 163

Sniffing out User Experience Smells

Jose Cuevas
jcuevas@ea.com

Abstract

It's a common misconception amongst developers that they know or understand what users want or need. It is equally a misconception that users know or understand what they want or need. However, what users want or need is much easier to identify once they actually use the product or service. So how do you design a product or service without being able to accurately determine the requirements prior to releasing it?

"…when you have eliminated the impossible, whatever remains, however improbable, must be the truth", Sherlock Holmes (Doyle 1890, 111).

Like a great sleuth faced with the daunting task of solving an impossibly complex mystery, you must eliminate the different layers of misconception to reveal what undoubtedly must be the truth. There are many clues, also known as "smells", which can lead you to the truth about your design. However, these smells can only be sensed by those who are willing to evaluate their own product or service with an open mind and a proactive approach.

There are two ways to identify smells:

1. Study and learn material on subjects related to the user experience.
2. Test and observe actual users performing tasks with your product or service.

It has been my experience that testing and observation is not very effective if you haven't spent time gaining an appropriate level of knowledge with user experience design. I have also discovered that explaining poor design is usually met with resistance and disbelief by developers. However, seeing is believing and developers should eat their own dog food once in a while. It's not my goal to discuss a lot of technical jargon, but to instead guide the audience of this presentation on an interactive journey as we explore the user experience through the eyes of the customer and sniff out user experience smells.

Biography

Jose Cuevas is a software engineer at Electronic Arts (EA) and is currently working at the Tiburon studio in Orlando, Florida. Since 2005, he has been involved in QA and Test Engineering for multiple games and currently serves as a production support specialist and developer of tools and services for automated testing of EA Sports games.

Prior to joining EA, Jose spent almost ten years in the Navy as a Nuclear Electrician, Radiological Controls Technician, and Nuclear Instructor. Jose also spent two years at the University of Advancing Technology in the Game Design program prior to leaving the Navy to pursue his dream of working at a major game company.

1 Introduction

The User Experience (UX) is that feeling in the back of your mind that moves to the front when you're either engaged with or discouraged by a given product or service. More importantly, it's based on an individual's emotional response to a given situation and not a collective perception on how the world should operate to meet someone's needs. I'm always fascinated by the developer that has spent long periods of time in a dark hidden corner of the office designing this amazing application to provide some really cool feature only to discover that it's nowhere near what anyone wants or someone else has already created something similar.

The lesson learned is that feedback must be collected as early and often as possible if you want to create something that truly meets the needs of the user and can adapt to changes in requirements (which will undoubtedly happen once the user actually uses your product). In addition, early feedback can avoid wasting time developing something that won't even be used or can be provided by a less expensive alternative through another vendor.

However, feedback is only good if value is obtained from it. It is my primary objective to ensure that I provide a solid set of material that is easy to follow and understand and can be used to improve any team's ability make good use of the feedback they collect.

2 Understanding the User Experience

The foundation of any subject is knowledge. If you do not understand the material that is being presented, you will miss the point. Before testing, it is extremely important to know who the users are and what constitutes a user experience.

2.1 History of the User Experience

Every so often, man creates something that allows us to perform the impossible. In the early phase of that creation's adoption, the design is only good enough to be useful. When computers were first created, it brought with it new possibilities. However, there was no focus on making it the best experience for users, only to provide users with the ability to do something they could never do before.

As the hardware and software evolved, so did the user's expectation of using them. Once users accepted the possibilities of what a computer can accomplish, the next logical step was to make accomplishing things more efficient. Once that was accomplished, the next phase was to make the overall process more pleasant (an emotional response). It is here that the User Experience was discovered.

2.2 Who are the Users

Before we discuss the experience, let's get to know the users. In short, the user is the human consumer of the product or service. So why have an entire section on users? I have two very good reasons:

- The end-user is *not* the only user of a product or service that matters.
- To make a great product, you should focus on your target audience and not try to please all people.

2.2.1 The Other Users

A major product or service is made up many smaller components. To identify the other users, it's important to identify the owners and stakeholders for each of these components. There are usually several internal products or services used within a company to either improve the development process

or be consumed to complete functionality. The end-user may never be exposed to these details, but an internal developer will. There is a cost associated with creating a tool or service that does not provide a good experience for the developer. Yes, developers are very smart people that can craft amazing solutions to very complex problems. However, that shouldn't be an excuse to avoid spending time designing the tool or service to be easier and less painful to use or implement.

2.2.2 The Right Audience

It's also extremely important to identify the right audience for your tests. Everyone is different; however, your target audience will have common goals and requirements that can affect their experience. Too wide an audience can result in feedback that may have little or no relevance to the target audience. If your product has something to do with racing cars, then your target audience should have some interest in competing in a race.

However, it's good to obtain feedback from outside the target audience range when:

- Trying to appeal to a more casual user or new user.
- Having difficulty getting testers within your target audience. Any feedback is better than none and will still reveal issues that your target audience may experience too.
- When trying to identify a new target audience.

2.3 Define the User Experience

Have you ever been asked to perform a task or job that you have very little knowledge in? The first thing you do is look up whatever information you can find on how to perform the task you have been assigned. However, teams are okay with performing tests without ever providing any type of training for the observers in regards to how to identify issues. Then they go into the testing debrief and dispute what problems are actually valid and which ones take priority over the others. Before I begin providing the quick and dirty on how to identify user experience problems, I want to throw out a definition from Wikipedia that will be the foundation for everything else that we will be presented in the following sections.

> ***User experience (UX)*** *is the way a person feels about using a product, system or service. User experience highlights the experiential, affective, meaningful and valuable aspects of human-computer interaction and product ownership, but it also includes a person's perceptions of the practical aspects such as utility, ease of use and efficiency of the system. User experience is subjective in nature, because it is about an individual's feelings and thoughts about the system. User experience is dynamic, because it changes over time as the circumstances change.* (Wikipedia 2012)

I will also add that the user experience is observed at any stage, multiple stages, or across all stages in the lifecycle of a given product or service. To summarize, the user experience is an emotional response by an individual to your product or service. Therefore, identifying issues requires knowing the individual on a personal level.

2.4 Company Experts

Depending on your company, you may have a team of people that are experts in the user experience. However, these experts are usually relegated to handling high level user experience problems (those dealing directly with your end users). Low level, or internal, user experience problems usually have no formal team that is responsible for their design and testing. Therefore, many internal tools and services suffer from very poor user experience problems which can have significant costs to your company because of the amount of time wasted by employees trying to use them. To avoid this pitfall, it is

important to train your developers on how to create a great user experience for your internal customers as well as the external customers and to also perform usability tests in a similar, if not the same, manner.

3 Top Ten User Experience Problems

An easy solution to getting quality feedback is by observing someone actually using the product or service. The best approach I have encountered for obtaining good feedback is to physically be a part of the testing process and get as many developers as possible to participate. There are a couple of things to keep in mind with this process.

Take a step back and discuss with your team on what things to look for (the smells of a bad experience). Some will be obvious because the user relayed the problem directly to you and others will be subtle and require some form of user experience awareness. I found Steve Krug's suggestions for usability testing in the books *Don't Make Me Think! A Common Sense Approach to Web Usability*, *Second Edition* (Krug 2006) and *Rocket Surgery Made Easy: The Do-It-Yourself Guide to Finding and Fixing Usability* (Krug 2010) the most helpful at obtaining user experience feedback. However, the feedback is helpful only if the developers understand why the user experience is poor. Many developers believe that the users think the same as them, which would completely invalidate the need for the term *target audience*. As a consolidated guide of information from multiple sources (Krug 2006. Krug 2012. Microsoft 2010. Saffer 2010. Cooper, James, and Saffer 2010. Garret 2010. Hollis 2012), I present to you my top ten smells of a bad user experience and suggestions on how to fix them:

3.1 Overwhelming

3.1.1 Smell

Users either have a specific reason or general idea about why they want to use your product or service. Research has shown that users typically scan for relevant content as opposed to analyze (Krug 2006, 22). The easier it is to scan, the more likely it will be for them to get what they need. However, if a user is overwhelmed by the amount of choices available to them, research has also shown that users will just choose the first option that seems to make the most sense (Krug 2006, 24). Frustration will settle in if the user has to endure multiple bad decisions because the right decision isn't obvious.

3.1.2 Fix

1. Simplify. Show the most relevant content for the target audience and remove any noise.
2. Size. Research has shown that users tend to gravitate towards the bigger objects as opposed to the smaller object (Hollis 2012). Make the most important decisions bigger than the less important ones.
3. Placement.
    a. Using conventions helps users feel comfortable knowing where to find things, such as the shopping cart in the top right corner of a website.
    b. Location is important because users tend to scan a page in a particular pattern. People who normally read left to right will scan from top-left to bottom-right. People who read right to left scan from top right to bottom left. Important content should be within those regions.

3.2 Distracting

3.2.1 Smell

Competing features will prevent users from focusing on relevant information. Imagine you're in a stadium trying to find your friends who are in the seated in the stands. If everyone in the stadium is jumping up

and down and waving their arms, it becomes very difficult to scan for your friends. However, it becomes much easier to find your friends when there's a break in the action and the only people standing up and waving are them.

3.2.2 Fix

1. Limit moving parts. If there are too many moving parts, the users won't know what to focus their attention to. If every object has an animation for example, it has the same effect as a stadium crowd going wild.
2. Use color wisely. If everyone in a room was wearing gray and one person was wearing red, the person wearing red would stand out. Wise use of color can help you guide the user to certain options and make other options fade to the background.

3.3 Annoying

3.3.1 Smell

Hearing or seeing the same thing over and over again can be annoying. Hearing or seeing the same thing over and over again can be annoying. Hearing or…I think you get the picture. Another annoying trait is constantly disrupting the user with noises or useless information.

3.3.2 Fix

1. Don't ask questions you already know the answer to.
2. Don't repeat warnings if the issue never cleared and provide an option to dismiss the warning.
3. Don't interrupt the user with needless information or sounds.
4. Use sounds sparingly. Also, make them configurable so users can turn it off if desired.

3.4 Confusing

3.4.1 Smell

This is an easy smell. Do the users know what they are doing? If not, assume they have been let down and there must be something we can do as developers to improve their decisions making process. Don't assume they are dumb, though it's quite possible, but instead assume that we are not clear.

3.4.2 Fix

1. Limit the need for training. A well designed product or service can be learned on the fly. Incorporate the learning into the doing. Try to avoid making the learning a separate process because most likely it will be skipped and/or ignored. Remember, most users scan.
2. Obvious signs. Like the yellow brick road in the Wizard of Oz, make the directions to the Emerald City a clear path. Use good default values and communicate clearly where to go and how things work. Lead the user down the right path. As Steve Krug wrote, *Don't Make Me Think* (Krug 2006).
3. Empower users. If there is a problem, show users how it can be resolved without assistance from support.
4. Find support. If support is needed, make sure it is obvious who to contact and how.

3.5 Poor Feedback

3.5.1 Smell

Does the feedback provide any actionable value or is it just unnecessary info that is arbitrarily presented at any given time? The purpose of feedback is communication but it should be relevant information to improve the experience. Hungry customers most likely won't appreciate a long talk with the cashier if they know it will delay them getting their meal. Also, make sure the feedback you provide makes sense to the user and not the developer.

3.5.2 Fix

1. Provide feedback when there is a problem that limits or stops progress and when there is something actionable to be performed. Limit feedback that is nonessential information.
2. How would your frontline support team respond to a user if asked a question? That's how the product or service should respond. Developer information should be captured but not presented to the user. Collect that data to be accessed by developers later.

3.6 Constant Stranger

3.6.1 Smell

When you get to know someone, you eventually stop having to asking them who they are and what they like. Your ability to communicate with those people becomes more efficient over time. Users like to be treated the same way. Don't let your product or service act like a stranger every time it communicates with the user. Also, get a feel for what they like and find ways to improve their experience by suggesting changes or features that may be beneficial.

3.6.2 Fix

1. Adapt to User. Get to know the user to personalize the experience. Don't provide a lot of options that requires the user to configure the experience because it is less personal and can lead to user errors (i.e. bad experiences). Learn about their usage and automatically modify the experience to fit their needs.

3.7 Time Consuming

3.7.1 Smell

I last time I went to the local Department of Motor Vehicles (DMV) office, I scheduled an appointment online. When I got there, they told me I had to wait in line. When I got to the line, they told me it was too long and I had to sit down. This is why I hate going to the DMV. I spent most of my time waiting and not enough time doing. Users want to get or do things, not wait.

3.7.2 Fix

1. Consolidate. If there are a lot of steps for a user to complete, try to find a way to consolidate them.
2. Automate. Complete information for the user that can be calculated by or is a repeat of already provided data.
3. Reminders. If the process is going to take a long time, give the user the option to walk away for a while and remind them when the process is complete.

3.8 Frustrating

3.8.1 Smell

Read the blogs, listen to the feedback, and take them seriously. If you're doing a user test, observe their facial expressions, body language, or even spoken language to identify their level of frustration. You'll know when the user is fed up, however, you'll want to identify all the little things that may have led to the meltdown and not just the last thing the user did.

3.8.2 Fix

1. Gather feedback. Observe, listen, learn, and then correct.
2. Pay attention to details. Small things matter and users will appreciate you taking time to make the subtle changes that improve their experience. Also, be aware of how your product or service is perceived by users with disabilities such as color blindness.
3. Limit negative experiences. A lot of little problems will equal one big problem. Every little mistake is an excuse to leave. Give them enough excuses, they may never come back.

3.9 Unresponsive

3.9.1 Smell

Nobody likes to get a deer-in-the-headlights look from the product or service they are using. They want to know that something, even if it means telling them that nothing is happening.

3.9.2 Fix

1. Feedback. If there is a long running process occurring, then give some type of indication that there is something going on and, if possible, how much longer it will take.
2. Perception. If the loading icon is moving really slow, the obvious assumption is that the loading is moving slowing. Additionally, if it is moving fast, then the loading is moving fast. If it is always moving fast regardless of how fast the actual process is moving, the user will feel much more comfortable. It's like going to a restaurant and waiting a long time for your food and all the waitresses are just standing around. I want to see some action like they're extremely busy and that's why my food is taking so long!
3. Distraction. If you're going to make them wait, this is a great opportunity to throw some advertisements or useful information their way. They do this in the movie theatres now well in advance of the first preview.

3.10 Fails

3.10.1 Smell

Poor functionality is a user experience killer. The product or service needs to work as expected first and foremost. I've seen teams spend way too much time discussing how to improve the aesthetics of the view when the functionality is failing. Provide value then improve usability.

3.10.2 Fix

1. Focus on what the users want most and make them work great. Don't add new features until those important features have been cleaned up and verified to be working great. Users will be forgiving of bugs in your least important features if the main features work great.

2. Expect the unexpected. Be prepared for server downtime, patches, etc. As developers, we need to be pessimistic about our own product or services. It's marketing's job to be optimistic.
3. Test your product or service. This should be obvious, yet we still have to say it because it's still a problem.

4 Conclusion

The user experience is a combination of many different components that drive one thing, how a user feels about your product or service. Testing the user experience is becoming a common practice and we are constantly studying new concepts and experimenting with different techniques to identify problems and create better designs. However, it is important that developers are well educated in the definitions and basic principles that fall under the user experience umbrella to ensure that the improvement process runs smoothly and your team or company achieves the most success with your product or service. I have provided a brief list of what I feel are the top ten issues and how to fix them based on information from multiple sources. Like a good sleuth, you can use this knowledge as a tool to help solve the mysteries behind many user experience problems.

References

Doyle, Arthur Conan. 1890. *The Sign of the Four*. London: Lippincott's Monthly Magazine.

Wikipedia. 2012. "User experience." Last modified July 1. http://en.wikipedia.org/wiki/User_experience.

Microsoft. 2010. *Windows User Experience Interaction Design Guidelines*. Last modified September 29. http://msdn.microsoft.com/en-us/library/windows/desktop/aa511258.aspx.

Krug, Steve. 2006. *Don't Make Me Think! A Common Sense Approach to Web Usability*, *Second Edition.* Berkeley: New Riders.

Krug, Steve. 2010. *Rocket Surgery Made Easy. The Do-It-Yourself Guide to Finding and Fixing Usability Problems.* Berkeley: New Riders.

Saffer, Dan. 2010. *Designing for Interaction, Second Edition: Creating Innovative Applications and Device.* Berkeley: New Riders.

Cooper, Alan, Reimann, Robert, and David Cronin. 2007. *About Face 3: The Essentials of Interaction Design*. Indianapolis: Wiley.

Garrett, Jesse James. 2010. *The Elements of User Experience: User-Centered Design for the Web and Beyond (2nd Edition).* Berkeley: New Riders.

Hollis, Billy. 2012. *Creating User Experiences: Fundamental Design Principles*. Pluralsight.com.

How Design Trade-Offs Influence Software Testing

Anil Chakravarthy (anil_chakravarthy@mcafee.com)
Pramod Sharma (pramod_sharma@mcafee.com)
Sudeep Das (sudeep_das@mcafee.com)

Abstract:

While designing and implementing software, architects and implementers make many different design and implementation choices. These choices reflect the trade-offs they make in order to solve a software engineering problem. Understanding those trade-offs is important to ensure that testing is planned and executed effectively. A trade-off implies that one attribute of the software is sacrificed to strengthen another. Testers need to make sure to treat the strong and weak areas, such as performance, availability, maintainability, security etc. in appropriate ways. Understanding trade-offs helps testers use correct assumptions about the software behaviour. It also helps them to set right level of expectations about the software behaviour.

In this paper, we look at the various design and implementation trade-offs made by architects and implementers. To illustrate our point, we use real world examples of the products that we worked on. We then suggest how such trade-offs should be handled effectively to make the testing more effective.

Biography:

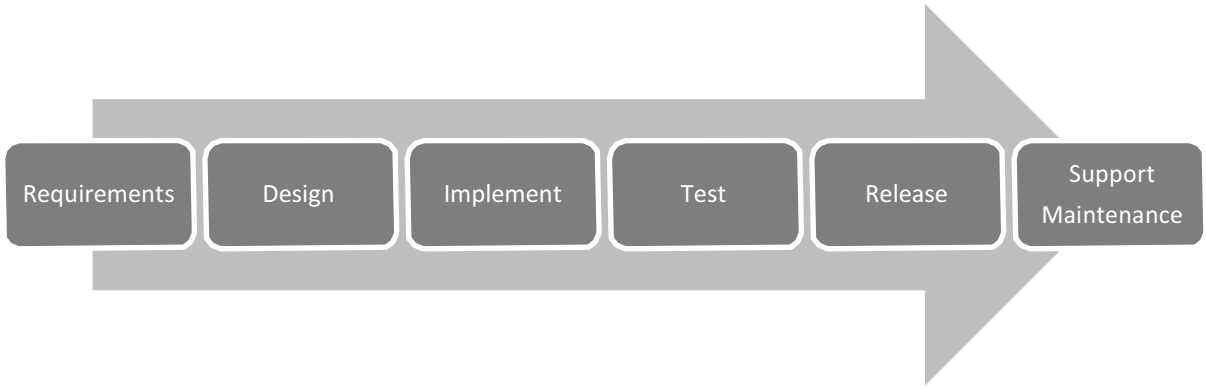
Anil Chakravarthy is a Technical Lead at McAfee, with more than seven years of software development experience. Highly passionate about quality, he strives to lookout for ways and methods to improve software reliability and usability. As an inventor of key technology solutions, his interests include security management, content distribution and updating.

Pramod Sharma is a Technical Specialist at McAfee, with more than eight years of industry experience. An acknowledged technologist, Pramod has eight patent pending inventions to his credit. Passionate about software quality and process improvements, Pramod is a vocal champion of adopting changes, and learning from the industry. His interests span security management for embedded and mobile devices, and scalable architectures for distributed systems.

Sudeep Das is a Software Architect at McAfee, with more than ten years of experience designing and implementing software. He started with a three person team and over time has seen it to grow to more than twenty, while improving upon development and testing processes incrementally over the years. His areas of interest span security management, content updating mechanisms, data protection and virtualization.

1. Introduction:

In general, a software development life cycle consists of requirements gathering, design, implementation, testing, release, support and maintenance phases. Each of these functions has significant impact on the quality of the software and on user experience.



Much of current software engineering discipline is focused on getting each of these functions right. It is also important to understand the influence of each of these functions on one another.

Each of these functions involves different sets of people with different areas of expertise. People involved in these functions need to understand the needs and limitations of people involved with other functions. For example, designers, while choosing a technology, need to ensure that the implementation team has sufficient expertise on the technology. Implementers need to ensure that the software allows for white box and automation hooks where necessary.

High bandwidth communication between the software engineering functions is key to getting the functional inter-dependencies right. The communication needs to be relevant and right information needs to be exchanged. In general, such communication is found to be between functions that are logically closely related. For example, communication between architects and implementers, between implementers and testers, between testers and support engineers etc.

However, it is also important for *non-immediate functional groups* to communicate effectively[1]. For example, testers and architects need to communicate effectively. This is one area where communication seems to be lacking. While the need to communicate is well understood, what exactly to talk about isn't. That's one reason why tester's participation in design discussions and architect's participation in test planning are often seen as mere formalities.

To demonstrate this shortcoming, and how it can be overcome, we take the example of one key aspect of software design, namely design trade-offs.

In this paper, we start with a high level view of software design trade-offs to familiarise the reader with the concept. We then take a look at some well-known software design trade-offs, and how they influence testing. We also look at the quality consequences that may occur when testers do not take into account these trade-offs. We conclude with some additional insights and recommendations to quality practitioners who may want to try out the discussed design trade-offs in their new projects

Our Intention is to get people to think about all aspects of software design, think about how those aspects influence software testing, and finally, think about what relevant aspects should form the basis for communication with the testing group. We also would like testers to think about how fore knowledge of various design facets can help plan and execute tests more effectively.

[1] Functional groups that immediately precede or succeed each other in a development cycle are said to be immediate functional groups. For example, implementation and design are immediate functional groups, but design and testing are not.

We believe this paper is to be of primary interest to architects, test planners, and project managers. However, all stakeholders in the software development process would find this paper beneficial.

2. What is a Design Trade-Off?

Architecture is the blue print of software. Depending on the constraints and operational environment of the software, various choices about the software's functionality, about the implementations, and about capabilities are made by architects. Requirements and constraints on the software force an architect to, at times, settle for a solution that gets the task accomplished but may not create the best possible outcome. Architects may also try to strike a balance to reconcile conflicting requirements. Such a balancing act sacrifices and weakens one aspect of the software to strengthen another. This is what we refer to as a design trade-off.

3. Trade Offs and Their Implications:

In this section, we list out the actual design trade-offs that software architects make. For each trade off we list here, we first provide a brief description of it. We then describe the test plan implications and test execution implications of the trade-off. We also provide a brief guideline on good practices that can help handle the testing impact of the trade-off effectively. We finally support the point with a real world example.

3.1. Robustness vs. Size:

Software, while operating, will run into error conditions. Robust software should handle such errors, do its best to recover from them and let the user continue on with the task. How well the software handles errors and recovers from them is an indicator of its robustness.

3.1.1. The Trade-off:

Robustness requires error handling and recovery. Error handling and recovery needs additional code to be written and additional code translates to additional binary size. When constrained for size, architects may sacrifice error recovery to achieve the size requirements. One way is to adopt the "fail fast fail safely" strategy, where any condition other than the "happy path" causes a controlled, handled but unrecoverable failure.

3.1.2. Testing Implications:

Before planning the test cases, testers should clarify with the architects if this design trade-off exists. They should also check with the requirements team about such a trade-off being acceptable and if so, to what extent. At times, requirements and error handling behaviour may need to drill down into details of what is acceptable and what is not.

Without this three way communication between architects, testers and requirements team, and with incorrect assumptions about error recovery, testers will raise significant number of defects for "as designed" cases. A sudden spike in the number of defects alarms management stakeholders about software quality. The concern is aggravated when those defects are marked "As designed". Considerable amount of time and resources are expended in handling the lack of communication.

3.1.3. Example:

We had to create a web installer for our product, a small application that would be downloaded by the end user and then executed to download and install the rest of the product. The web installer had severe size constraints, and so, for various error conditions, the design and implementation simply

printed an error, and exited.

There was lack of early communication between testers, architects and product manager on this aspect. The testing expectation was to have the installer recover from errors and allow users to continue. This was contrary to observed behaviour and caused debates, arguments and escalations about the product behaviour.

3.2. Make vs. Buy:

3.2.1. The Trade-off:

Architects, at times, may choose to use third party code or libraries instead of spending significant resources on creating supporting technology from scratch. Frequently, this third party code is open source. The flipside of this is that the software now has code on which the development team may not have expertise. The third party code may have its own share of defects that the development team may be ill equipped to fix.

3.2.2. Testing Implications:

Testing team members should clarify upfront the usage of third party code. If there is third party code involved, then all stakeholders should agree on the following:

a. Treatment of defects encountered in third party code:

   The possible options are

   a. Agree to fix by the development team: This is not recommended in practice unless the development team has expertise on third party code.
   b. Fix by third party code providers: This is a feasible option, and is highly recommended. There is a risk that some of the third party code may not be fixed by the time the software needs to be released.
   c. Ignore: This is not recommended as this implies deliberately shipping buggy software.

b. Adjustment of code coverage metrics:

   The possible options are:

   a. Completely ignore: In code analysis and coverage metrics, completely ignore any metrics from third party code from coverage reports. If the third party code is known to be robust, stable and well regarded, then this is the recommended approach. This is so because certain well regarded third-party open source components like Boost (C++ libraries), libcurl (URL transfer library) etc. follow well established development processes and are known to be mature and stable. As such, there isn't much gain in establishing coverage metrics for such code.

   b. Adjust for functionality: For not so mature third party components, it may help to map the parts in third code that will actually be used by the product being developed, and include them in the coverage metrics. While somewhat difficult in practice, a recommended way is as follows:

      Consider a third party i/o library that has exposes a read() and a write() function. Let's say the read() internally calls read_usb() , read_sd(), and write() internally calls write_sd() and write_usb().

      i. Identify all API (Application Programming Interfaces) in the third party code that are ACTUALLY used by the product being developed. In this example, it is read() that will actually be used.
      ii. Evaluate a code flow graph of only those APIs of the above step in third party code.

This will provide all functions and classes that will be in the code path of the API of third party code. In this example, the flow graph for read will traverse read_usb() and read_sd(), but will not touch the write functions.

iii. Map code corresponding to only those third party functional code in coverage metrics.. For this example, you will include only code for the read, read_sd(), and read_usb() in your metrics

Not being aware of this trade off and not planning for it may have undesirable consequences as below:

Inaccurate and misleading code coverage metrics: The application may be using only a small part of the third party library, and hence only that part will be covered in coverage metric. For example, if a program with 1000 LOC (Lines of Code) uses a third party component with another 1000 LOC, but exercises only 200 lines in there, then coverage metrics will indicate only 60% coverage or 1200 out of 2000 total LOC.

Misleading static analysis metrics: Static analysis applications like Coverity and Fortify will analyse the third party code as well, and will indicate problems with them.

3.2.3. Example:

In one of our products, there was need for significant usage of a particular compression algorithm and http(s) capabilities. Building them by ourselves would have been at a significant cost, and longer time to market, so a decision was made to use third party components that, in addition to what we needed, provided a lot of other capabilities that we didn't need. Code coverage and static analysis raised red flags after analysis, and considerable time was spent later to tailor the results to reflect the true state of the code.

3.3. Feature Parity vs. Platform Disparity:

3.3.1. The Trade-off:

In many instances, the same software may need to be supported on multiple platforms and operating environments. Different operating environments and systems have different capabilities. Depending on the feature set of the software and the platforms it is expected to be operational on, software designers may choose to:

a. Support a feature set that is limited but uniform across all platforms, based on lowest common denominator of platform capabilities.

b. Add platform specific code to support different environments.

3.3.2. Testing Implications:

Product managers, architects and other stakeholders should evolve consensus on handling platform disparities. From a testing perspective, it's a common error to create test cases for one platform and then extend it to all platforms. Testers then test the features on platforms that don't support it, and then create defects for them. This gets repeated for each additional platform that the software is supported on.

Testing professionals should not assume feature parity across platforms. While creating test

cases, they need to ensure that feature disparities across platforms are taken into account. If this is not done, then midway through the development cycle, test cases will need to be revisited, altered and re-executed to account for platform disparities. Planning for this trade-off upfront is important.

3.3.3. Example:

McAfee's products support multiple platforms. For example, on Windows, a feature set exists that is tied to getting OS notifications about user dial up connection. Not having a reliable way to do this on Linux, Solaris, AIX and HPUX the feature was not available on these platforms. This caused significant user experience and consistency concerns to the testing team who assumed feature parity across platforms.

3.4. Security vs. Usability

3.4.1. The Trade-off:

Different software systems have different security needs. For example an apartment doesn't need Fort Knox security, and prison system cannot depend on padlock security.

Although more security checks improve security assurance, they may negatively impact user experience by making it harder to use the system. The degradation may be seen in performance for example, when dealing with large amounts of cryptography, and in user experience where a user may be forced to establish credentials at every operation. Given that security can never be fool proof, designers can only raise the bar. Both Security and usability are competing requirements, designing software with right balance based on acceptable levels is the key trade-off.

3.4.2. Testing Implications:

Testers responsible for security testing should discuss this aspect with architects before planning their testing. The discussion should also include product managers to ensure that the level of security and usability is in line with user expectations.

Security testing tools should be configured based on the security design of the product. Testers should exercise reasonable judgement on the scope of security testing, rather than running all possible exploit tools on the product.

Similarly, testers validating usability need to be aware of the security aspects that may limit usability, and plan and execute their tests accordingly.

3.4.3. Example:

One of McAfee's products is an enterprise client server product, but is an on-premises solution where both client and server are inside the enterprise network boundary. As such a few security measures relevant to internet facing systems were dropped, because the server was not me ant to be internet facing. Because of this misunderstanding the security testing team ran a significant number of tests in that area, and came up with defects that were not applicable to the situation.

3.5. Memory vs. Performance:

3.5.1. The Trade-off:

Another classic trade-off is that performance drops if less memory is available and improves with

more memory. Depending on the resource profile of the target system, designers may choose algorithms that are either resource intensive or resource constrained.

3.5.2. Testing Implications:

Performance testers should discuss this aspect with architects and product managers before planning their performance testing. Depending on requirements, architects may replace entire algorithms while keeping functionality intact. This may cause performance metrics to be outside expected bounds. By communicating with architects about this aspect, performance testers can ensure that their performance testing metrics are an accurate reflection of application performance.

3.5.3. Example:

We had a product that was originally intended for a general purpose computing environment, and used a resource intensive algorithm to deliver performance. In a subsequent version, the product was required to support mobile devices as well, and the algorithms were changed to resource conservative ones. The product in the newer version had a resource conservative algorithm across the board. When the general purpose compute environment was taken up for performance testing, the performance results were below the previous version benchmarks. The performance testing team raised slow performance related issues, as a result time and effort was spent to discover the source of the problem. After consultation with the product manager, it turned out that a slow performance was expected because of new the algorithms. The testing team was unaware of this design change, tested against the original benchmarks, leading to significant waste of resources.

4. Conclusion:

Effective communication across different functions is fundamental and a key success factor in delivering quality software. Tri-partite communication between architects, testers and product owners (managers) is one such example. The effectiveness of such communication depends on the right set of questions being raised and the right set of problems being addressed. The purpose of such communication is not to make architects out of testers or testers out of architects. The purpose is to encourage early collaboration, exchange of information about the roles and responsibilities of each of the functions, how it is intended to be done, and how it may affect other functions.

We believe that the various trade-offs presented here will encourage architects to discuss in-depth the trade-offs during their interaction with testing teams. We also believe that testers will actively ask questions about various design choices instead of being observers in design meetings. There are many more design trade-offs than the ones discussed here we hope software professionals add to this list over time from their experiences

Knowledge of design trade-offs helps testers to plan software test activities wisely. It sets the correct expectation about what the software is intended to do. Mismatched expectations on software behaviour causes significant time to be lost in unproductive testing activities, and may also require significant rework.

A case study into improving Network device driver quality by design

Anjali Singhai Jain
Kevin Scott
anjali.singhai@intel.com
kevin.c.scott@intel.com

Abstract

In this paper we discuss some of the key software design decisions which were made to facilitate driver development and validation prior to receiving silicon. We discuss engineering collaboration across the hardware, software and firmware teams as partners in the generation of driver code, as well as using driver synthesis to generate code.

The goal of the project was three fold: 1) Develop drivers with a significant number of code paths which are easily exercisable. 2) Allow both open source and closed source drivers to share resources and obtain higher quality code as a result. 3) Allow as much code as possible to be Open Source.

We discuss the lessons learned on what and how to share between drivers, and how to develop the code such that it can be used seamlessly by driver teams with various needs (Linux, Windows, FCoE, iSCSI, DCB, RDMA, silicon firmware modules, etc.).

Biography

Kevin Scott is a Network Software Engineer who has been with Intel's LAN Access Division for 12 years.

Anjali Singhai Jain is a Network software Engineer at Intel Inc., currently working at the Intel site in Portland, Oregon. Over the past 8 years, she has been involved in software development for tools, early prototype libraries and network device drivers for Intel's Networking division. Anjali as part of tools team has mastered in software design for validation and writing OS agnostic modules. She has been actively involved in using her learning's in her current position as Linux network device driver developer.

Anjali has an M.S. in Computer Science from Portland State University.

1 Introduction

In general, as computing systems have advanced, the sophistication and complexity of the software and hardware that comprise them has increased. This general trend is mirrored in Intel's network controllers. As feature lists in new hardware grows, the ability to write the various pieces of software which control, validate and test the hardware become more complex. This increase in complexity also generally means that development times (Time to Market) is greater. Methods to reduce development time for a new product with uncompromised quality must be identified.

This paper discusses one such code development mandate, which we call "Shift Left". The Shift Left mandate is a corporate wide effort to address time to market challenges.

We will focus this paper on the following road-blocks to our Shift Left goals:

- How to increase the scope of sharing code across platforms, environments, and teams/functionality (Firmware, Hardware, Silicon Validation, Software).
- Splitting Hotpath (code traversed repeatedly during packet processing) and non-Hotpath code and keeping the later common.
- Designing an OS abstraction layer to allow more code to be shared.
- Auto-generation of hardware definitions for use by drivers and tools i.e. driver synthesis.
- Co-development with other software groups.
- Ability to validate software paths prior to FPGA arrival.
- Software device emulation.

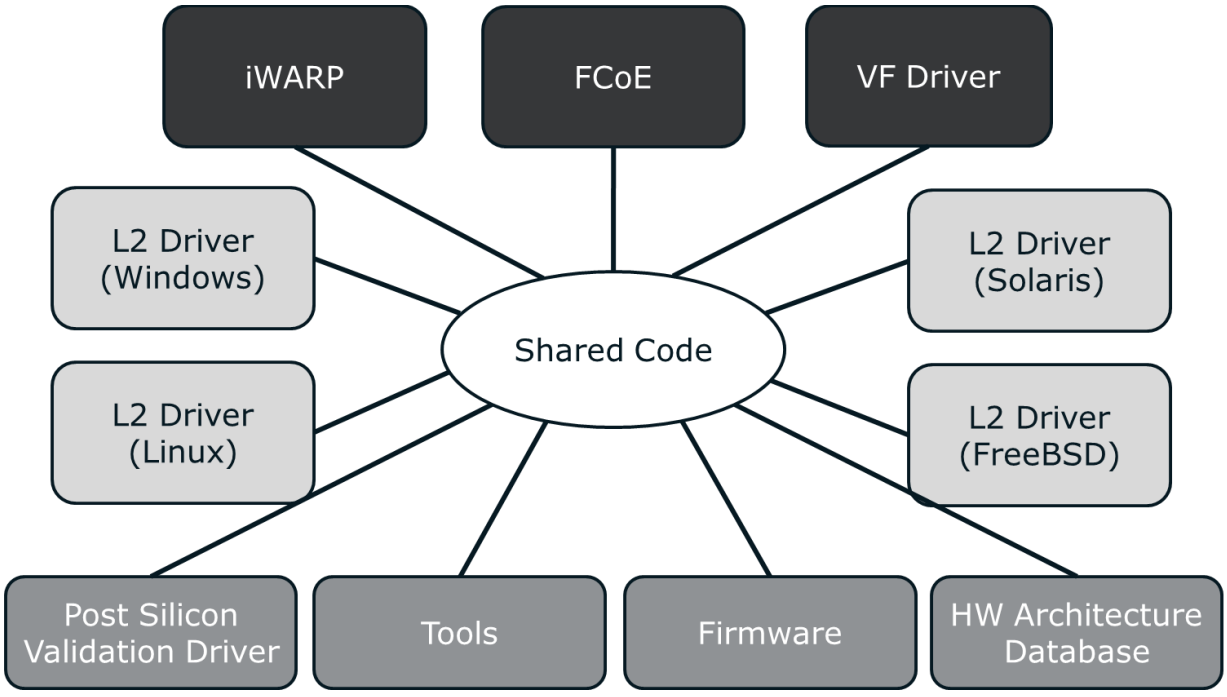


Figure : Various components that share code.

Figure 1 shows the different drivers, tools and components that share huge chunks of code. This was made possible due to early shared code design, keeping the consumer needs in mind and identifying the OS abstraction APIs needed for the same.

2 Design Philosophy

Intel has adopted the Shift Left mandate for its LAN products in an effort to release products in a timely manner; by moving project milestones earlier (to the left) in their lifetime. As silicon complexity has grown, due to supporting technologies such as SR-IOV (Single Root I/O Virtualization), DCB (Data Center Bridging), FCoE (Fibre Channel over Ethernet) and RDMA (Remote Direct Memory Access) among others, the overall product development timeline has remained the same or even shrunk.

Additionally, in order to remain competitive in the market, it is required that the time between the arrival of first silicon to the time that silicon is available to customers, is as short as possible. This effectively mandates that much of the software development and validation occurs prior to first silicon arrival.

The Shift Left mandate has introduced new complexities in the product definition / software life cycle model:

- The hardware architecture specification may not be complete prior to the beginning of software development. There are numerous “details” which are not fully understood or fleshed out at this stage, so how can the software respond to this challenge?
- Different software components will be in development at the same time ex: firmware and OS base drivers. This forces components which are being actively developed at the same time to function together at some level.
- The software has to be validated on something prior to FPGA (Field Programmable Gate Array) arrival. Since the FPGA may arrive close to the arrival of first silicon, this leaves little room for validation post FPGA arrival.

This single mandate has forced teams to rethink, redesign and innovate how they develop software. Converting some serial tasks into parallel tasks and identify new partners to indirectly implement and validate their drivers.

Intel LAN software has historically utilized shared code, source code which is used by two or more components, to develop various drivers and tools. The amount of code which was shared varied, as some teams needed to develop their components before the shared code was functionally complete. The time constraints, as viewed by individual teams, would lead to purpose built code and ultimately, a duplication of effort.

In order to address this, the shared code had to start development earlier, grow in scope and be used by a wider developer audience.

2.1 Automatic Code Generation

One method to allow for earlier code development is to combine the act of defining hardware specifications by the architecture teams and the generation of code which can be used to manipulate the definition i.e. the auto-generation of code.

Previous Intel LAN products have used little, if any auto-generated code. All definitions for register offsets, masks, shifts, etc. were manually coded by a developer, who obtained the information from an Engineering Architecture Specification (EAS). The EAS in turn is written by a team of silicon and software architects, over a long period of time. During this time, things such as register definitions are modified and lead to errors in the shared code.

The Intel Networking Group is now using the tool which architects use to define the silicon (RTL), and develop the EAS, to generate code which is directly useable in the shared code. This all but eliminates

one of the historically more problematic issues of code development; the manual task of translating an EAS into shared code.

Another benefit of auto-generated code is the consistency of style. Since the shared code is part of the Linux kernel, avoiding submission issues because of simple style guideline errors can greatly speed upstream submission. On projects with many developers, and varying experience with the style guidelines, auto-generated code makes a lot of sense.

For reference, the Linux base driver for a future networking product contains over 13,000 lines of auto-generated code and roughly 60,000 lines of total driver code. So, currently, approximately 25% of the base is auto-generated. If this amount of code were generated manually, it would have resulted in many individual code patches, and the overhead associated with the generation and validation of such patches. This is a significant savings in time and effort.

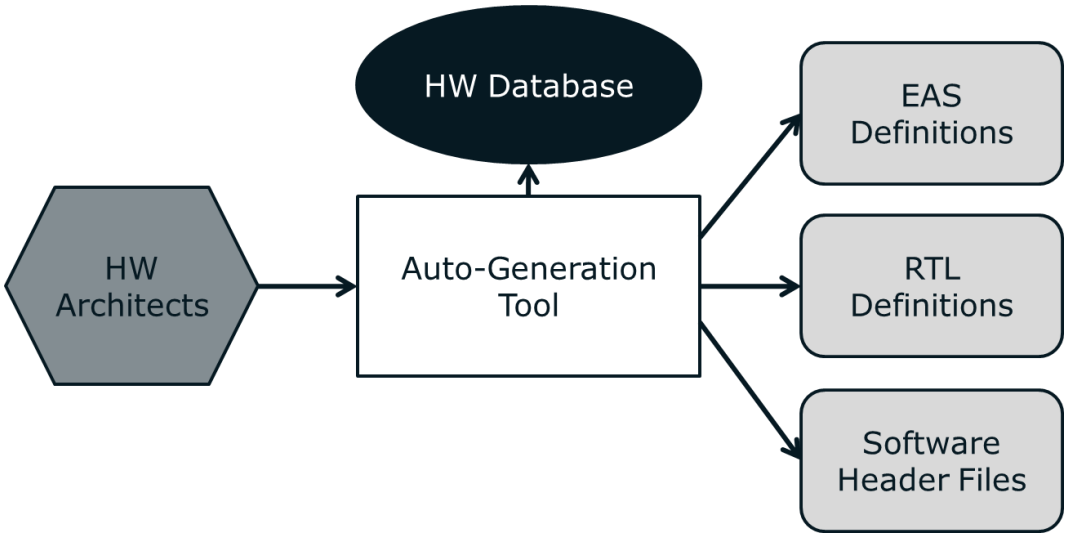


Figure : Auto Code Generation

Figure 2 describes the actions of the code auto generation tool. The tool is used by the hardware architects to define the contents of the definitions database. The tool uses this single database to produce content for the EAS, C header files, RTL code etc. The tool also tracks revision history and can be used across various generation of products.

2.2 Code Re-Use

Code duplication has historically been a problem, as different components must meet widely different timeline requirements. For example, in order to test and validate FPGAs destined for software developers, the silicon validation team would write a custom driver (SV driver), which by its nature was similar to a high level OS driver (Linux or Windows), yet the code developed could have been used in the shared code.

With the Shift Left mandate, developers across disciplines agree to work with a single code base, derived from the architect's specification, and this code in turn is used in numerous tools and drivers.

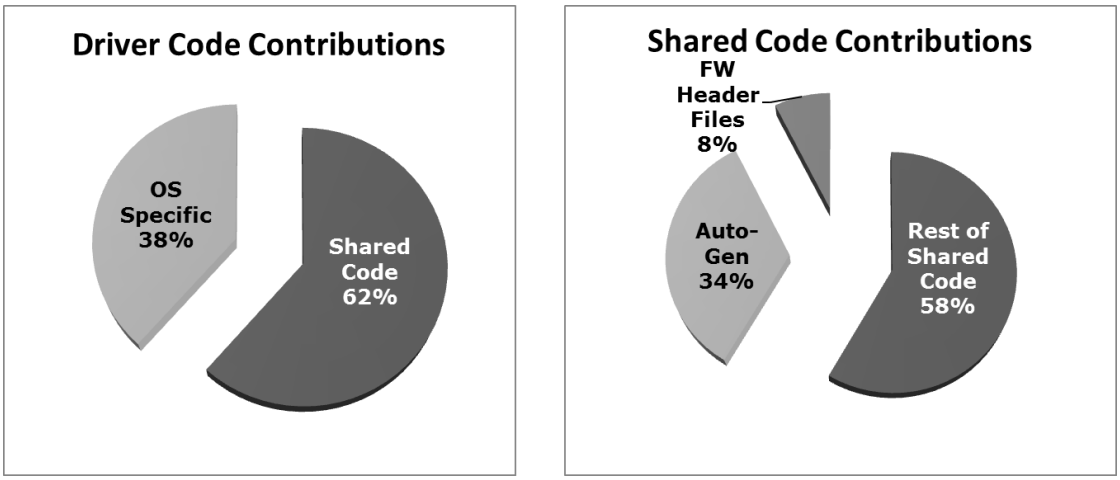


Figure : Shared code pieces vs. OS specific code in Linux Device driver.

Figure 3 shows the percentage split of different contributors to the Linux device driver. Left chart shows percentage of Total shared code vs. OS specific code. Right chart shows what percentage of shared code consists of Auto-generated code, FW code and the rest. Shared code between OS drivers, validation drivers and Tools has become a major portion of the overall device driver.

In order for code-reuse to have a meaningful impact, we focused on two major areas:

- OS abstraction APIs to aid with keeping code common
- Identifying what should be part of common code and what should remain in OS specific files

2.2.1 OS Abstraction

Historically, the shared code could be considered a very thin abstraction layer between software and hardware, in that it consisted of fundamental hardware/software interface mechanisms. To increase the role of shared code, we have abstracted roles which were normally handled by OS specific drivers:

- Allocation of both physical and virtual memory structures
- Memory manipulation functions
- Code Synchronization: spinlocks
- Delays
- Debug mechanisms (debug prints and driver logging mechanisms)

Here is an example of a set of OS Abstraction APIs as declared in the shared code. The definitions are filled out by each of the OS drivers in the OS specific files. This is a good example to illustrate how we designed the APIs to meet various drivers' needs to allocate the memory from different pools depending on its use. Another driver could chose to completely ignore the type and allocate from a common pool.

Code Abstraction Example:

```
/* Memory allocation types */
enum i40e_memory_type {
        i40e_mem_arq_buf = 0,          /* ARQ indirect command buffer */
        i40e_mem_asq_buf = 1,
        i40e_mem_atq_buf = 2,          /* ATQ indirect command buffer */
        i40e_mem_arq_ring = 3,             /* ARQ descriptor ring */
        i40e_mem_atq_ring = 4,             /* ATQ descriptor ring */
        i40e_mem_pd = 5,                   /* Page Descriptor */
        i40e_mem_bp = 6,                   /* Backing Page - 4KB */
```

```
        i40e_mem_bp_jumbo = 7,              /* Backing Page - > 4KB */
        i40e_mem_reserved
};
/* prototype for functions used for dynamic memory allocation */
enum i40e_status_code i40e_allocate_dma_mem(struct i40e_hw *hw, struct i40e_dma_mem
*mem,
                                            enum i40e_memory_type type, u64 size, u32 alignment);
enum i40e_status_code i40e_free_dma_mem(struct i40e_hw *hw, struct i40e_dma_mem *mem);
enum i40e_status_code i40e_allocate_virt_mem(struct i40e_hw *hw,  struct i40e_virt_mem
*mem, u32 size);
enum i40e_status_code i40e_free_virt_mem(struct i40e_hw *hw,  struct i40e_virt_mem *mem);
```

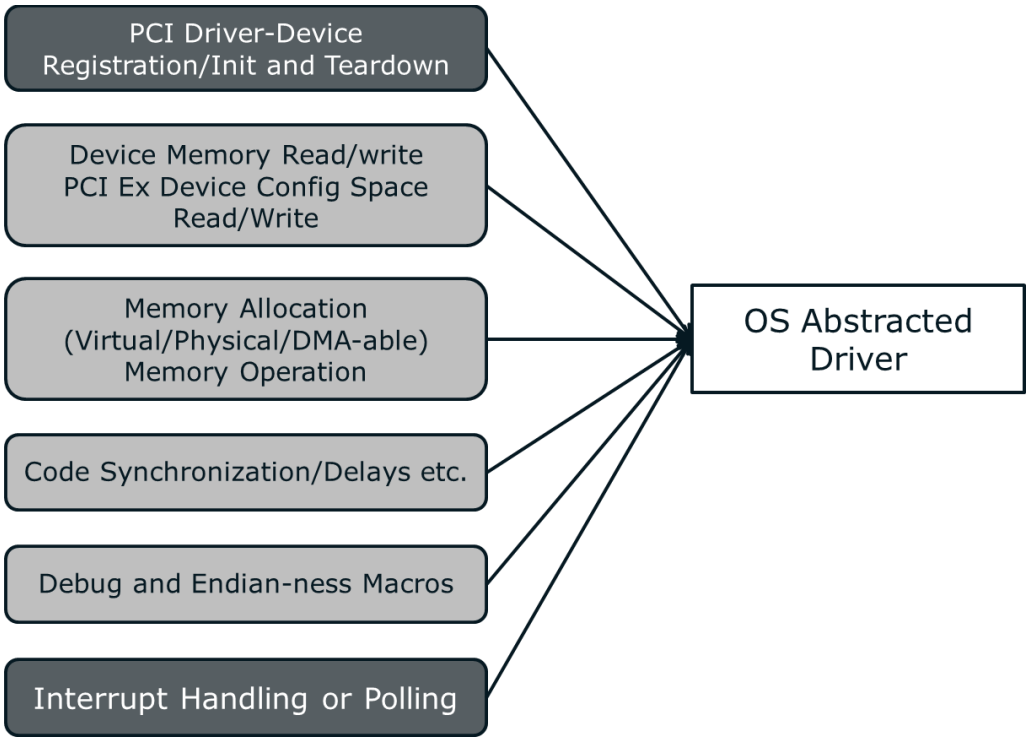


Figure : OS Abstraction parts.

Figure 4: This figure takes a leap into the future. Blocks in the middle are abstracted as of now in our driver. If we were to consider a completely OS abstracted driver for various reasons in the future, we would have to come up with OS abstraction APIs for the top and the bottom block.

2.2.2 Hotpath vs. Non-Hotpath

The shared code primarily focuses on code (Non-Hotpath) which is run outside of Interrupt Service Routines (ISRs) and Deferred Procedure Calls (DPCs) i.e. the Hotpath. We term the Interrupt or the DPC code as Hotpath since it is not desirable that the driver; these portions of the code is where all of the packet processing happens in the driver and any code here impacts the total throughput of the driver. Keeping Hotpath code out of shared code allows for OS specific drivers to tune it in the most efficient way to achieve relevant performance characteristics. On the other hand, Initialization codes, configuration code, tear down and reset code is non-Hotpath and is very Hardware focused, these make the best candidates for Shared code.

2.3 Co-development

The shared code is simply too complicated and large for any single team to develop. Historically, the various teams have worked and developed in a fairly compartmentalized manner, coming together and sharing details only when there is some sort of problem; often an emergency. Now, the insights and perspectives of the various teams have a single point of focus. Features which may initially be needed by only one team, and written in team specific code in the past, are now shared, so that when any team needs to use said feature, they simply have to incorporate what is already there. Features, workarounds, errata do not need to be re-discovered.

2.3.1 Firmware/Software team as a model for co-development

Similar to the hardware specific definitions being owned by the hardware team, the firmware definitions are owned by the firmware team and used by both the firmware and OS driver teams. Though not auto-generated, the firmware and software teams now use identical code (Shared Firmware Header files) to develop their separate drivers. Since there is a single owner (the firmware team), the code is always accurate with the latest specification, and can't be misinterpreted by the OS driver teams.

The firmware header files generation follows the below rules:

- Shared code style guideline.
- Patches generated solely by firmware team.
- Reviewed by software shared code team.
- Maintained by software shared code team.
- Consumers are both software and firmware teams.

3 Testing Scope

Shared code has historically been written with OS base drivers in mind, so the code paths which are developed and exercised are generally limited to operations only the base drivers would consider valid. With multiple teams developing the shared code, the scope of testing and validation is widened. Code paths which may not have been exercised previously are now exposed and testable.

3.1 Early validation partners and tools

3.1.1 Shared Code use by Post-Silicon Validation Drivers

Prior to the OS driver teams obtaining an FPGA or first silicon, the silicon validation teams validate hardware behavior on a silicon validation driver, which is based on the Linux driver. As this driver is developed with the shared code, very early feedback on the health of the code is obtained. In addition to this early feedback, the testing methodology of the silicon validation team is very different than that of the OS driver teams. The Silicon validation tests are heavily automated and randomly generated, designed to exercise various hardware boundaries and extremes; paths the OS driver code would only exercise in rare cases. This increases overall shared codes robustness.

3.1.2 Hardware emulation: KVM-QEMU

Historically, OS drivers could only be run once FPGA devices arrived and were programmed and tested by the silicon validation team. However, with the Shift Left mandate, this is no longer reasonable. In order to exercise OS driver code prior to FPGA, we have emulated our network device in software using KVM-QEMU framework provided with standard Linux kernel. Though this only emulates a small set of

total device features, it allows the OS drivers to exercise critical code paths: initialization, transmit, receive, DPC/interrupt, etc. It also makes possible testing other components of the driver stack, such as protocol drivers and user mode components which may leverage features in the base driver.

Advanced offloads and features are areas for improvement going forward.

4 Linux Specific Changes to Improve quality

In our software model, there are many different drivers that interact at various levels with the silicon.

4.1 Common Client Interface Model

Earlier versions of our base drivers and silicon had many different ways to interact with protocol drivers; some co-existed with the base driver, and others resided in their own modules using a proprietary interface. In our new driver model, we have devised a generic interface between the base driver and the protocol drivers. This helps reduce code, complexity and provides easy ways to scale and test.

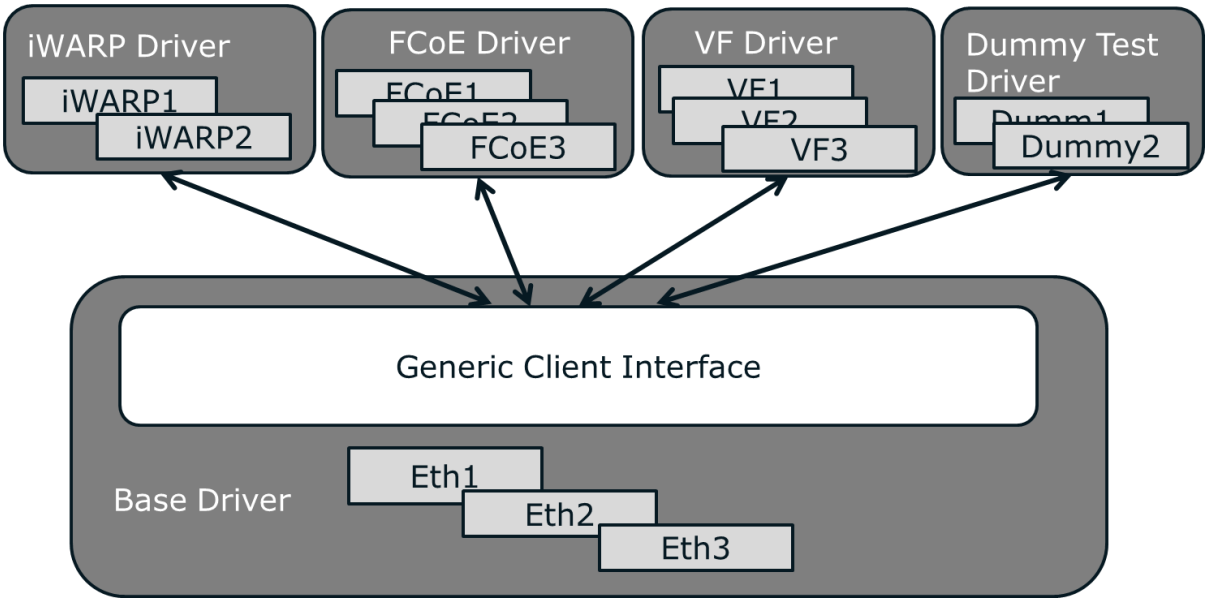


Figure : Client Interface Model in Linux

Figure 5 describes the Generic Client Interface that is implemented in the base driver and exposed among all protocol drivers that need the base driver services. Base driver exposes the various PCI devices present in the system that the driver supports as individual HW Interfaces (ex. eth0, eth1 etc.). The protocol (Client) drivers each in turn have instances corresponding to the base driver Interfaces. Each Client instance is tied to its respective base driver interface through the generic client Interface layer. iWARP is a name given to the RDMA driver in this case. VF drivers are the Virtual function drivers that use SR-IOV to boost VM Network performance.

4.2 debugfs

We now use the debugfs support built in Linux kernel in our driver for:

- Driver validation support during development and project releases.

- Driver configuration interface for developers/users/administrators.

Earlier, we would use custom debug macros in our driver, which were not very extensible. They did not provide for any easy configuration interface; module parameters and ethtool were our only choices for driver configuration. Module parameters are not very well received by upstream kernel development team and ethtool has its limitation in terms of what gets accepted as configuration knobs by the different Network silicon vendors that have Open source presence. debugfs in that respect can be programmed to expose anything and everything the driver writer would like exposed for configuration. This has tremendously increased validation scope for our Linux based drivers.

5 Conclusion

Though the development process is not yet complete for our first driver utilizing the development methods we have described is complete, we have been able to successfully address many of the challenges presented by the Shift Left mandate.

1) As technologies grow more complex and the boundaries between software/hardware/firmware blur, the ability to share as much code as possible between them becomes more important. Utilizing tools which allow for the automatic generation of code eliminates many 'lost in translation' errors which would be experienced by various teams in various geographies, and incur a high cost in time to debug, weakening the Shift Left mandate.
2) A small investment of time and effort in designing the right OS abstraction APIs goes a long way in enabling different OS driver teams to share large amounts of code, helping reduce individual team efforts, bugs and lines of code to be maintained. Choosing which code to share should be done prudently. Sharing code in the fast path limits the ability to make OS specific optimizations to the code, and can limit overall driver performance.
3) Early Hardware emulation in software, by driver developers, helps greatly. It provides a mechanism to validate code paths ahead of silicon. By writing code to the hardware specification, software teams can help identify difficult areas to manage or code for. Driver developers also develop a greater understanding of the hardware model they are developing for. Above all else, we were able to find and fix hundreds of driver bugs prior to silicon arrival. We found that this ability alone justified the time required to develop a hardware model.
4) A common abstracted client interface for advanced technologies (RDMA, FCoE, and Virtualization) provided for scalability and code-reuse. This model provides for building blocks that plug on top of base driver in a generic fashion hence providing for easy testing.

6 Future Directions

The current level of automatic code generation is still limited in scope; it covers only register offsets (and ranges), mask and shift definitions. Extending code generation to structure definitions, field values and names (enumerated types or defines) will further reduce the time it takes to develop code and reduce chances for error. This would approximately double the size of present auto generate code in the drivers.

The OS abstraction layer in shared code in the future could allow for simplified driver models with very little OS dependent code. These would be non-performance diagnostic drivers whose goals would be to exercise code flows and features which are not easily tested in the standard OS environment. Another consideration is the development of a driver which is forward compatible, allowing network connectivity without a full featured OS driver. Such a driver could be deployed in the market prior to silicon, enabling the new silicon out of the box for basic functionality at all times.

7 Glossary

iSCSI (Internet Small Computer System Interface) - iSCSI is a protocol to transport SCSI commands over Internet Protocol (IP) based networks, generally used for storage connectivity.

FCoE (Fibre Channel over Ethernet) – Protocol which encapsulates Fibre Channel packets on standard Ethernet based networks.

FPGA (Field Programmable Gate Array) – An integrated circuit which can be repeatedly programmed and modified.

DCB (Data Center Bridging) – A set of Ethernet extensions to help facilitate functionality in data center environments.

SR-IOV (Single Root I/O Virtualization) – A mechanism to allow a Single Root Function (ex: an Ethernet port) to be exposed as multiple physical functions.

RDMA (Remote Direct Memory Access) – A mechanism to allow bus mastering devices (ex: Ethernet controllers) to place data directly into user space, rather than to kernel memory and then copied to user space.

debugfs – Linux file system used for debugging purposes. Allows for a convenient kernel to user space communication mechanism.

EAS (Engineering Architecture Specification) – Document which describes the technical details, specifications, and interfaces of a particular piece of silicon.

SV (Silicon Validation) – Process by which silicon behavior (post production) is compared to expected behavior, as defined, in part, by an EAS.

DPC (Deferred Procedure Call) – Function which processes required work for the Ethernet device outside of the interrupt context.

RTL (Register Transfer Language) – A low level language used to describe the behavior of operations in integrated circuits.

Shared Code – A single code base which is used by multiple software components to help reduce the amount of code duplication.

KVM-QEMU –Kernel-based Virtual Machine- Quick Emulator

Self-Verifying Data

Douglas Hoffman
Software Quality Methods, LLC.
Doug.Hoffman@acm.org

Abstract

Some tests require large data sets. The data may be database records, financial information, communications data packets, or a host of others. The data may be used directly as input for a test or it may be pre-populated data as background records. Self-verifying data (SVD) is a powerful approach to generating large volumes of information in a way that can be checked for integrity. This paper describes three methods for generating SVD, two of which can be easily used for large data sets.

For example, a test may have a prerequisite that the database contains 10,000,000 customers and 100,000,000 sales orders. The test might check adding new customers and orders into the existing data, but not directly reference the preset data at all. How can we generate that kind of volume of data and still be able to check whether adding customers and orders might erroneously modify existing records? SVD is a powerful, proven approach to facilitate such checks.

The paper and talk describe the concepts, applications, and methods for generating such data and checking for data corruption. They cover:

What self-verifying data is
Why and how self-verifying data can be used
Applications where such data is useful
Three ways to apply self-verifying data
How to check the data records generated this way

Biography

Douglas Hoffman BACS, MSEE, MBA, ASQ Fellow, ASQ-CSQE, ASQ-CMQ/OE

Doug has more than thirty years of experience with software engineering and quality assurance. Today he teaches and does management consulting in strategic and tactical planning for software quality. Training specialties include context-driven software testing, test oracles, and test automation design. His technical specialties include test oracles, test planning, automation planning, and developing test architectures. Management specialties include ROI based planning of software engineering mechanisms, QA management, organizational assessment, evaluating and planning for organizational change, managing and creating project management offices, building new quality organizations, and transforming existing quality assurance and testing groups to fulfill corporate visions.

Doug is very active professionally: President of the professional society Association for Software Testing, ASQ Fellow, Senior member of ACM and IEEE, past Chair of the Silicon Valley Section of ASQ, past Chair of the Silicon Valley Software Quality Association, Chair and organizer of numerous quality conferences, over 50 conference presentations, and written dozens of publications on software testing, quality assurance, and management. He holds degrees in Computer Science, Electrical Engineering, and an MBA. He also has ASQ certifications in Software Quality Engineering and Management of Quality/Organizational Excellence.

1 Introduction

One view of running tests is observing the software under test (SUT) for proper behavior given some stimulus. The stimulus comes from many sources, direct input and referenced data among them. The amount and nature of the required data varies based upon the requirements of the software under test and the purpose of the test. Although not always the case, good tests should generally be repeatable, start from a known state, and have verifiable results (performance tests, race condition tests, and multi-threaded asynchronous tests being some counter examples). Therefore, these tests require data that is repeatable and will result in verifiable outcomes when processed by the SUT.

Tests sometimes require large volumes of inputs (e.g., run-time generation of millions or billions of values) and huge data sets (e.g., pre-populated databases) that can result in difficulties in verification of test outcomes and determining of test verdicts. Self-verifying data (SVD) can answer questions like “did any other records get corrupted?” when a test is complete. Detection of reasonable or unexpected software behavior is an oracle problem that is compounded with large volumes of data. Self-verifying-data is self-descriptive data, i.e., the data contains the key or clues as to what the data is supposed to be. For example, “This sentence is 52 characters in length (including spaces).”

SVD is not useful for all test situations. Sometimes the SUT has no requirement for pre-populated data (e.g., computations in a calculator) or there may be no place for a key (e.g., uid/password combinations). The SUT’s processing of data may be too complex to be determined with a single key (e.g., image analysis). But, SVD can be a powerful way to tag and track expected outcome checking.

Although data generation and outcome verification may be integrated within a test, for simplicity in this paper I will logically separate data collection/generation from running the test on the SUT from the verification of outcomes. Whether the data is generated before or during a test is not important to the concept of SVD, nor is it important whether the verification of outcomes occurs during or after the SUT is run. Logically separating data generation, SUT execution, and verification allows us to examine SVD more clearly.

Some notation and terminology:

CRC	A Cyclic Redundancy Check is a computed numeric value that represents a set of data. Any changes in the data will result in computing a different CRC value
Oracle	A test oracle is the principle or mechanism by which we recognize potential erroneous SUT behavior (e.g., comparison with expected results)
RAND	A random number
RNG	Random number generator
Seed	A number used to initialize a RNG
SUT	Software under test
SVD	Self-Verifying Data

1.1 Data generation

Data for SVD can be generated by a human or by a computer. Generally speaking, once the concept of SVD is understood, a human can be very creative in its application. However, the volume of human generated data is severely limited in comparison with computer generated data (unless you consider the human writing programs to generate the data). In this paper I lean heavily toward computer generated data that is useful in interesting automated tests.

Actual generation of well-formed data is generally straightforward. The rules for a grammar can be defined and data generated following the rules. The trick in SVD is embedding a key with the generated

data for later verification. The key does not have to be unique but it needs to have very high probability of matching the data to be very useful. For example, a computed 32 bit cyclic redundancy check (a.k.a. CRC) always matches with the data used to generate it but will randomly match other data 0.000000025% of the time.)

1.1.1 Random numbers

Randomly generated data is a useful mechanism for augmenting the specific test values needed for testing. Boundary conditions and other special cases provide the foundation for functional testing, while random values can provide tests for hidden special cases or generation of filler data. There are several ways to select or generate large volumes of test data. When we generate high volumes of SVD we generally employ random number generators (RNG) to select or construct the data. There are a few important characteristics of computer random numbers that are particularly relevant for this process.

- Computer random numbers are *pseudo*-random numbers. They exhibit statistical randomness but are generated using repeatable algorithms. I.e., a large set of random numbers have the statistical characteristics of truly random numbers, but the specific values form a repeating sequence that cycles after an astronomical number of generated values.

- If an initial value (called a *Seed* value) for the random sequence is specified, the same sequence of random numbers will be generated. If no *Seed* value is given, the RNG chooses one using highly unpredictable sources (like the system clock).

Using a seed value we can repeat a series of random numbers. This means we can reliably rerun a test or data generation that uses random numbers.[1] If we want a new random sequence we can use the random number generator to create a new *Seed* value. The RUBY code in Figure 1 shows an example of code using repeatable random sequences.

```
MAX_SEED = 1_000_000_000

 def initial_RNG_seed(myseed)

    if (myseed == nil) # Check if seed value is provided
       # Create a random number to seed RNG
       Puts "(no seed passed in, so generate one)"
       myseed = srand() # start the random sequence randomly
       myseed = rand(MAX_SEED)  # generate a new Seed value
    end

    # Save the seed so that we KNOW what seed we used
    puts "myseed is #{myseed.to_s}\n"
    foo2 = srand(myseed) # initialize the RNG
    foo = rand() # generate the [first] random number
    return foo
end
```

Figure 1: Ruby code for generating or reusing random number seeds

[1] Some care is required for reliable repetition or random sequences. For example, asynchronous threads and programmatic variable scope changes are difficult to manage.

Figure 2 shows sample output of the seed and first random value from invoking the initialization routine in Figure 1. If a *Seed* value is provided it is used to initiate the same random sequence (shown in the second and fifth cases). If no *Seed* value is provided the program generates and logs a new randomly generated *Seed*.

```
puts ("First run: #{initial_RNG_seed(nil)} \n \n")
puts ("Second run: #{initial_RNG_seed(400)} \n \n")
puts ("Third run: #{initial_RNG_seed(nil)} \n")
```

→

(no seed passed in, so generate one)
myseed is 144288918
First run: 0.3705579466087263

myseed is 400
Second run: 0.6687289088341747

(no seed passed in, so generate one)
myseed is 108495905
Third run: 0.09838898989988143

```
puts ("Fourth run: #{initial_RNG_seed(nil)} \n \n")
puts ("Fifth run: #{initial_RNG_seed(400)} \n \n")
puts ("Sixth run: #{initial_RNG_seed(nil)} \n")
```

→

(no seed passed in, so generate one)
myseed is 173770139
Fourth run: 0.5201072369109297

myseed is 400
Fifth run: 0.6687289088341747

(no seed passed in, so generate one)
myseed is 320781144
Sixth run: 0.020652681349145108

Figure 2: Example output from running the Ruby code in Figure 1

1.1.2 Timing of record generation

The test data may be generated independently before the SUT is exercised or as part of the test. Populating large data sets is done before running the test. Randomly generating data communication packets as input to the SUT is done within the test. Although designed and applied somewhat differently, the timing of data generation is unimportant for the purpose of SVD, since the SVD mechanisms are the same either way.

1.2 Data verification

The purpose of generating SVD is so the data itself contains clues as to what the data should be. As described below, there are several SVD approaches and techniques. Though the oracle techniques applicable to each type of SVD are somewhat different, the common thread is that the data provides information for verification of test outcomes.

2 Three mechanisms for SVD

SVD can be generated in three ways: self-descriptive, cyclic algorithms, and random generation. Each is described below.

2.1 Self-descriptive data

We can create data that describes its own attributes. 20 point text, there is a box around these characters, and *underlined italicized Arial text* are some examples. This approach is primarily used in human (manual) testing although it is possible to programmatically generate such data. For self-descriptive data the interpretation of the data becomes the source for the oracle, which is extremely difficult to automate, as discussed in Section 3.1.

2.2 Cyclic data

Test data may be generated using a pattern of repetition. An initial value is created and additional values concatenated based on duplication or a simple repetitive algorithm. The key(s) are prepended or appended to the data. As described in Section 3.2, the oracle is then implemented by repeating the data generation from the key and comparing data, or by identifying the pattern and thus deriving and comparing the key(s).

2.2.1 Repeated values

The simplest method is generating text or arithmetic values by repeatedly concatenating data. The pattern key can be fully defined with two values: the starting value and number of iterations. This works equally well for text or numeric data.

For example, “ab10abababababababababab” and {55, 10, 55, 55, 55, 55, 55, 55, 55, 55, 55, 55} are repeating series. Cyclic data of this type usually has the key prepended when a generation oracle is applied and added at the end when pattern recognition is used. (See Oracle mechanisms in Section 3.2.)

Some care is required with text patterns to avoid ambiguous patterns, especially with manually generated sequences. Although highly unlikely with randomly generated data, the possibility exists. The data portion of “ab10abababababababababab” is the same as “abab5abababababababababab” and “abababababababababab1abababababababababab,” so there are three different ways this pattern could be interpreted and a key derived from the generated data.

2.2.2 Extended numeric sequence

An extended numeric sequence is a series of different numbers based on very simple algorithms. The algorithm generates each sequential value using a simple transformation of the previous value. Addition

of a constant is the most common method. The pattern key can be fully defined with three values: starting value, constant, and count of numbers.[2]

One example of an extended numeric sequence is a numeric sequence where each number is the result of adding a constant to the previous number ($X_n = X_{n-1} + K$, where K is a constant)[3]. The set {5, 7, 10, 5, 12, 19, 26, 33, 40, 47, 54, 61, 68} is an example where the key is {5, 7, 10} (start with 5 and add 7 for 10 values) and {5, 12, 19, 26, 33, 40, 47, 54, 61, 68} are the values. With the numeric values concatenated with the key values, checking can be done using either oracle method described in Section 3.2.

2.3 Random data

Probably the most powerful mechanism for generating SVD is using random numbers. Although self-descriptive and cyclic data can be generated randomly, random SVD depends only on the use of the same RNG and Seed values. The key used in random SVD is the Seed value. The seed is either embedded within the data or otherwise associated with the records as described in Sections 2.3.4 and 2.3.5.

Random SVD can be generated and used in several ways as described in Sections 2.3.1 through 2.3.3.

2.3.1 Nonsense padding

For this case data is collected or generated for space-filling purposes without expectation that any interpretation or processing of the records will occur. The values generated are place holders used to increase the amount of data and are passive for the purposes of the test(s). The SVD in this case is used as verification that the data has not become corrupted. This type of SVD is often used for performance and stress tests.

Because the data is not specifically processed, the values do not need to be meaningful. It only needs to be well formed enough to not throw exceptions. For example, names and addresses do not need to be meaningful, but only acceptable characters should be in the fields. The address, for example, does not need to have a real city's name.

For example, we may generate a million random users to assess an application with a large user base. These users are added to the set of specific users needed by the basic functional tests. None of the randomly generated users are accessed because each test uses their own specific users. Yet, all tests may include the larger data set for robustness.

2.3.2 Well-formed random records

For this case data is generated in well-formed records that may or may not be processed. A grammar is created describing the record and it is then applied using random values. Figure 3 shows a RUBY program to generate simple well-formed arithmetic statements using a hard-coded grammar.[4] (The generated statements are randomly created without seed values for simplicity.) Note that separate grammars may be required for each data field and data constraints among fields can make generation very complex.

---

[2] The cyclic numeric and text SVD mechanisms were used to test randomly generated data packets sent across a multi-modal network that included some half-duplex links (Ethernet, radio, power lines, and four other types of networks in series) where returning the data for analysis was impractical.

[3] Note that numeric overflows may occur, in which case the value can be truncated and the sequence continued.

[4] Although difficult to achieve, this type of generation of well-formed data can be generalized by reading in a grammar specification, usually for some group of data formats. For example, specifying different math functions or data base records.

```
def eq_gen
    jubilee = 1 + rand(8) # Matches the 4 cases we have defined
    case jubilee
      when 1
          "(" << eq_gen() << ") + (" << eq_gen() << ")"
      when 2
         "(" << eq_gen() << ") - (" << eq_gen() << ")"
      when 3
        "(" << eq_gen() << ") * (" << eq_gen() << ")"
      when 4
         "(" << eq_gen() << ") / (" << eq_gen() << ")"
      else
        rand(100).to_s  # 50% chance of generating a number
    end
end
```

Figure 3: Ruby code to generate arithmetic expressions

Figure 4 provides some examples of random well-formed equations generated by the program.

```
puts ("First:  #{eq_gen.to_s} \n\n")
puts ("Second:  #{eq_gen.to_s} \n\n")
puts ("Third:  #{eq_gen.to_s} \n\n")
puts ("Fourth:  #{eq_gen.to_s} \n")
```

→

First: (77) - ((62) / (6))

Second: ((10) - (40)) + (67)

Third: 53

Fourth: (62) - ((96) * ((((77) - (72)) - ((7) * ((47) - (91)))) / ((34) + (((70) - (18)) + (4)))))

Figure 4: Randomly generated expressions from Ruby code in Figure 3

2.3.3 Execution time random data generation

One useful way to use SVD is generating test values as part of a test. Either nonsense or well-formed SVD can be generated by the test as input to the SUT.[5] Nonsense data is likely to trigger error mechanisms in the SUT (which may be the intent of the test). Well-formed data is more likely to be processed by the SUT and the results of processing can be checked.

[5] This technique may be applied without appending the key, depending on the oracle mechanisms used. An example of execution time generation of well-formed input without a key is function equivalence testing (where the generated input is sent to the SUT and an alternate implementation such as different spreadsheets) and the returned results compared within the test. Since this does not require SVD, there is no further discussion of this mechanism.

The results from the SUT processing may be checked as the test progresses or through post processing the data. Checking as the test progresses may be done within the test or by a separate oracle mechanism receiving or monitoring the results.

2.3.4 Appending a seed value

Sometimes it is possible to simply append the seed to the test data. For example, the seed could be the first in a numeric sequence of random numbers. In the example below, the seed is appended to the randomly generated text.

For the example shown in Figure 5:

- Assume the seed (*S*) is 8 characters and name field holds a maximum of 128 characters
- To generate a random name:
  1. Generate a random number seed (S) and convert it to a string (assume 8 characters)
  2. Initialize the random number generator using SRAND(*S*) (or RAND(*S*))
  3. Generate a random Length (*L*) (up to 120 characters) using RAND()
  4. Generate a random name (*N*) with L characters using RAND()
  5. Concatenate the seed to the name
- Verify the name by extracting the seed and regenerating it

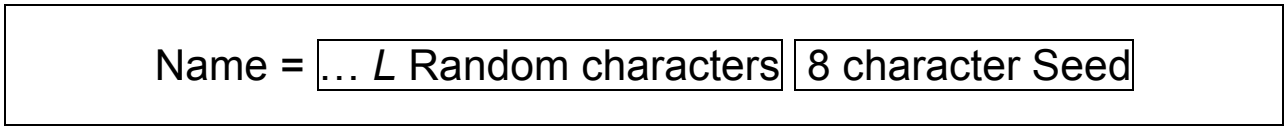


Figure 5: An example of embedding the seed within test data

Another example from Section 2.2.2 is the set {5, 7, 10, 5, 12, 19, 26, 33, 40, 47, 54, 61, 68}. Here the key is embedded as the first three values describing the cyclic series {5, 7, 10}, followed by the generated data {5, 12, 19, 26, 33, 40, 47, 54, 61, 68}. The ten data values can be easily generated starting with 5 and adding 7 each time. [3]

2.3.5 Adding a field for the seed value

Sometimes it is not possible to append the seed with the data. For example, date fields and binary values have no flexibility for added information. In these instances we need to add an associated field with the seed value. This is straightforward for database records by adding a new field, but can be tricky with data in say, a checkbook application.

For example:

To create a database record:
1. Generate a random number Seed (*S*)
2. Store the Seed value in an added field within the record
3. Generate the record using the Seed and an algorithm
4. Regenerate the records by applying the algorithm with each Seed

As mentioned, some data sets do not lend themselves to easily appending the seed values. Often there are unused fields or free form fields for notes where the seed can be inserted. Where there is no available field it may be necessary to create a separate table to contain the seeds. Each seed needs to be associated with its generated data, so some unique record identifier is needed. For example, in a

checkbook application we might create a table linking the check numbers or deposit amounts/dates with their corresponding seeds.

3 Oracle Mechanisms for SVD

As described in the introduction, the oracle is how we tell good SUT behavior from problematic behavior. An oracle can be built into the test, where the test periodically checks for expected results. It can also be built separately from the test, to be run during or after running the test. There is no limit on the number of oracles that can be created to check specific test outcomes or general program and system attributes. (A memory leak detector is an example of a separate check that runs in parallel with the regular tests.) The focus here is on the specific oracle mechanisms for SVD.

Depending on the application of the SVD, verification can happen during or after running the test (or both). SVD is typically not required when results are checked within a test because the expected outcomes being checked do not need to be reconstructed using a key. An advantage of execution-time checking is possibility of stopping to diagnose errors when they occur. Post-execution checking is a matter of reading the SVD and using the key to verify it. An advantage of post execution checking is exposing bugs for conditions we did not consider when the test was designed and implemented. However, post-execution checking can only identify that data was corrupted.

For example, the checkbook records can each be rechecked after test completion to detect unintended side effects. Where the test may check for specific results and expected potential side effects, there are an infinite number of ways the SUT may unexpectedly modify other data. Post execution checks are likely to uncover corrupted data from any other bugs or side effects not specifically checked in a test.

3.1 Oracle mechanisms for self-descriptive data

Self-descriptive data is usually hand-generated and uses a human as the oracle. Automatic generation of self-descriptive data is difficult, but can be achieved using the random well-formed data approach described in Section 2.3.2. Automating an oracle for randomly generated, self-descriptive data is very difficult because a key only provides a syntactical description of the data. Recognizing the actual attribute(s) of the data is a sematic problem. A human tester can read and interpret this kind of information more flexibly and reliably than a computer. However, checking whether the data has the specified attributes can be straightforward if they can be recognized.

For example, “20 point text” is self-descriptive and the size attribute can be randomly chosen. However, automating the recognition that it *is* 20 point text is nearly impossible, while a human can easily distinguish between that and “20 point text.”

3.2 Oracle mechanisms for cyclic data

There are two applicable oracle mechanisms for cyclic data: regeneration of the data and pattern identification. Either mechanism can be run during a test or afterwards.

Regeneration of the data is done by applying the same generation rules using the key values. The resulting data can then be compared. For large data sets, generation and comparison may be run without saving the generated data and in some situations it may be possible to generate and compare without keeping either the test or oracle data by generating and comparing expected results as the test runs.

Pattern identification is done by deriving the key value(s) by analysis of the data and then comparing keys. In the case of a data communications test the oracle was at one end of the chain of links the test was run at the other end. The test generated data was verified and the pass/fail result returned. For the

set {5, 7, 10, 5, 12, 19, 26, 33, 40, 47, 54, 61, 68}, the first data value is 5, the difference between values is 7, and there are 10 data values {5, 12, 19, 26, 33, 40, 47, 54, 61, 68}. Thus, the key is {5, 7, 10} (which matches the SVD key).

3.3 Oracle mechanisms for random data

The most practical test oracle is to apply the original data generation algorithm using the seed to regenerate the expected data. (Reverse engineering the seed is basically a decryption task with very little data for each seed, which is both time consuming and error prone.) The oracle can be as simple as reading the row from a table and using the key to regenerate the data for comparison, or quite complex; for example extracting the key from a note field to check the data extracted from a printed invoice for a randomly created sales order . The oracle usually uses the same program (or a variation of the program) to regenerate the data since the algorithm is established to generate that very data.

Post-processing is typical for randomly generated SVD, although there are cases where the expected results are checked as the test runs.[6] Post-processing has the advantage of checking all the records, while checking during a test run can catch problems early to make diagnosis easier. An independent oracle may also check SVD records continuously while tests are running (e.g., checking and rechecking database records).

4 Conclusions

SVD embeds a key within a set of data that describes the data itself. SVD is very useful for testing in many contexts because much of the software we test is stimulated using data that can be self-descriptive, and therefore checked for side effects. SVD can be self-descriptive, cyclic, or random. Self-descriptive SVD is usually generated and checked by humans. Cyclic SVD employs a repeating pattern that can be described with a few key values. Random SVD uses random numbers to create and tag test data. The trick with checking data is recognizing when the SUT does something unexpected (the oracle problem). SVD is one approach that facilitates checking outcomes because the key to the expected result is embedded within the data.

- SVD may be useful when:
  - The data lends itself to self-description
  - The data can be thought of as records
  - A key or seed can be used for data generation
  - Incorporation of the key with the data is straightforward
  - The test requires a high volume of inputs or referenced data
  - Checking for data corruption after test completion

- Not useful when:
  - The SUT does not use record type data
  - Checking data within a test
  - Outcomes don't reflect SVD type data records
  - The data structure is too complex
  - The data is not easily generated from a key
  - The key is not easy to include with the data

[6] Randomly generated data was also used in the data communications testing described in Footnote 2.

Shake 'n' Send: Enabling feedback submission directly from mobile applications

Billy Landowski
willand@microsoft.com

Sira Rao
sirarao@microsoft.com

Abstract

As the mobile application market continues to grow, product teams need effective ways to gather user feedback about their mobile applications in order to improve software quality. Although extensive application testing is performed by dedicated product test teams, it is extremely valuable to enable beta users outside of the product team to try the application and provide feedback on their experience. This provides the product team with real input from real users that closely align with target customers.

To solve this problem for mobile applications, the Microsoft Lync test team developed a tool called *Shake 'n' Send* to enable feedback submission directly from the Microsoft Lync mobile applications. This tool allowed beta users to provide feedback by simply shaking the phone and providing information about their experience using the application. The simple and easy-to-use interface enabled users to quickly report on product bugs and other quality issues. It also allowed them to "send a smile" about what they liked.

For Microsoft Lync mobile application beta testing, *Shake 'n' Send* has proven to be very effective. Using this tool, beta users submitted several thousand feedback reports and that information helped the Microsoft Lync Mobile team increase software quality by fixing an additional 30% more bugs in the product.

Shake 'n' Send has proven to be a fun and efficient way for beta users to submit feedback directly from mobile applications. Because of this success, the feedback tool is currently being embraced and extended to support other mobile product teams within Microsoft.

Biography

Billy Landowski is a Software Development Engineer in Test at Microsoft. For the past two years, he has worked in the Microsoft Lync group testing mobile applications. His passion is to create new ways of testing to have business impact, and have fun. Billy has degrees in Computer Engineering and Mathematics from Washington State University.

Sira Rao is a Test lead at Microsoft. He has worked on the Unified Communications products at Microsoft for over 8 years. He is passionate about building high quality products that excite customers.

1 Introduction

As the mobile application market continues to grow, product teams are finding an increasing need for methods to elicit feedback from users about their experiences with mobile applications during the product development cycle in order to improve software quality. For desktop applications, obtaining user feedback can easily be done through the use of additional buttons, websites, or completely separate applications because of the abundance of screen real estate, multitasking, and reliable network connectivity. However, mobile devices have a number of unique features that require a different approach to developing feedback mechanisms, including: constraints imposed by the OS, unreliable network connectivity, and significantly reduced screen real estate.

In this paper, we discuss the challenges of soliciting and collecting feedback from mobile application users, previously implemented solutions and their drawbacks, and a new tool that the Microsoft Lync test team designed, developed, and deployed to overcome these challenges that we called *Shake 'n' Send*. We also present results from using *Shake 'n' Send* with Microsoft Lync mobile applications and discuss how it has improved the quality of the software.

The following terms are used throughout the paper:

- **Dogfooding**: This term refers to the practice whereby product team members and others in a company actively use and identify issues in a product prior to release. This practice has been in use within Microsoft since 1988.
- **Microsoft Lync**: The next generation of Microsoft's unified communications software that enables people to connect in new ways, anytime, anywhere.
- ***Shake 'n' Send***: The tool that is central to this paper, helping to enable sending feedback from directly within a mobile application. The tool name is coined after the actions to initiate ('*Shake*') and submit the feedback ('*Send*').

2 Background

In this section, we discuss general challenges that software developers must face while developing on mobile platforms and how these challenges affect the "send feedback" process for mobile application product teams.

2.1 General challenges of mobile applications

Mobile applications by their nature are encumbered by a variety of challenges that desktop applications may take for granted. One of the most obvious differences between the mobile and desktop platforms is the screen size, most of which are smaller than five inches. This limitation imposes new design challenges for product teams. With so little real estate, every pixel must be carefully planned for in the user interface design, making it very costly to change the design once it is finalized.

For security purposes, most mobile platforms enforce strict "sandboxing" rules which limit applications to their own data, and most applications cannot communicate with each other. Similarly, the traditional concept of "multitasking" on the desktop has different implications on the mobile platform. Since platform resources such as computing power and shared memory are scarcer than desktop operating systems, most mobile platforms only allow one application to actively run in the foreground at a time. Although some applications can register with the operating system to run in the background, there is no guarantee that the application will be given sufficient resources to operate.

Network connectivity can also pose many problems for mobile application developers. Access to the internet is not always guaranteed, and developers must accommodate for this by planning for more failure

scenarios and adding additional error checks. Furthermore, developers cannot rely on access to internal corporate networks because mobile devices are designed to work independently of them.

2.2 Challenges of applying desktop feedback mechanisms on mobile applications

Many different feedback mechanisms exist for traditional desktop applications, but as we discuss in this section, applying these methods on mobile applications is no easy task.

2.2.1 "Send Feedback" Button

One standard method for submitting feedback from desktop applications is to add a static "Send Feedback" button to the application's main user interface. For example, in Figure 1 below, the "Collect Logs" button on Microsoft Lync 2010 has been added to the released product for users to submit their feedback directly to Microsoft. This method is quite easy and inexpensive for desktop applications, but due to the reduced screen size on mobile devices, the real estate for such a feedback button is extremely expensive.



Figure 1 - Microsoft Lync 2010 screenshots of the send feedback process. When users press the "Collect Logs" button (left), they are presented with a new dialog to give their feedback (right).

2.2.2 Separate application

Another option for sending feedback on desktop applications that developers can use is implementing a completely separate application for sending feedback. By doing so, developers can use the same "Send Feedback" application for a variety of products that the company may produce, and the end user will have a similar and familiar "Send Feedback" experience for all of that company's products. One may be able to imagine this as a solution for the mobile platform; however, if the "Send Feedback" application needs any sort of diagnostics or additional data from the mobile application that is being critiqued, there is no way to gather this information due to the "sandboxing" nature of the mobile platform.

2.2.3 Link to website

Many desktop applications have simply added a link to a website where users can submit their feedback on the product. This is a nice option because it requires minimal changes to the desktop application, plus

it can rely on access to the internet. This option is much more complicated for mobile devices, however, because internet access may not be available for capturing the user's feedback. Perhaps more importantly, using a web link would require navigating away from the application in order to submit the feedback, disrupting the user's interaction with the application.

3 *Shake 'n' Send* Solution

So far, we have only discussed the limitations of mobile applications when compared to desktop applications, but there are also many unique capabilities that mobile platforms provide as well. Some examples include: GPS, built-in cameras and microphones, and accelerometers.

In order to solve the "Send Feedback" problem for mobile applications, we decided that instead of looking at just the limitations of the mobile platform, we should also look at the benefits and see if we could utilize any of these unique features. Hence, after experimenting with the built-in accelerometer, we decided to pursue a solution with physical gestures, and more specifically, the "shake" gesture.

3.1 Using the "shake" gesture

In order to solve the small screen real estate issue on mobile devices, we discovered that we could remove the dependency on additional user interface elements by using the mobile device's built-in accelerometer. Instead of tapping on a "Send Feedback" button, the user can simply shake their device to initiate the feedback process. With this idea, we developed a tool called *Shake 'n' Send* that is built directly into the mobile application. This enables the user to access the tool from anywhere within the mobile application without the need to navigate to a website or separate application.

When the user wants to send feedback, they simply shake their mobile device (like shaking a can of spray paint). When the shake gesture is recognized, a new user interface is presented to the user on top of the existing application where the user can type and send their comments to the product team directly from the mobile application. Screenshots of the *Shake 'n' Send* user interface can be seen in Figure 2 below. Additional information such as diagnostic data and traces as well as application screenshots can be captured at the time of the shake and sent to the product team with the user's comments. After the information is sent, the user is returned to the mobile application right where they left off, providing a complete closed-loop experience for sending feedback from the device.

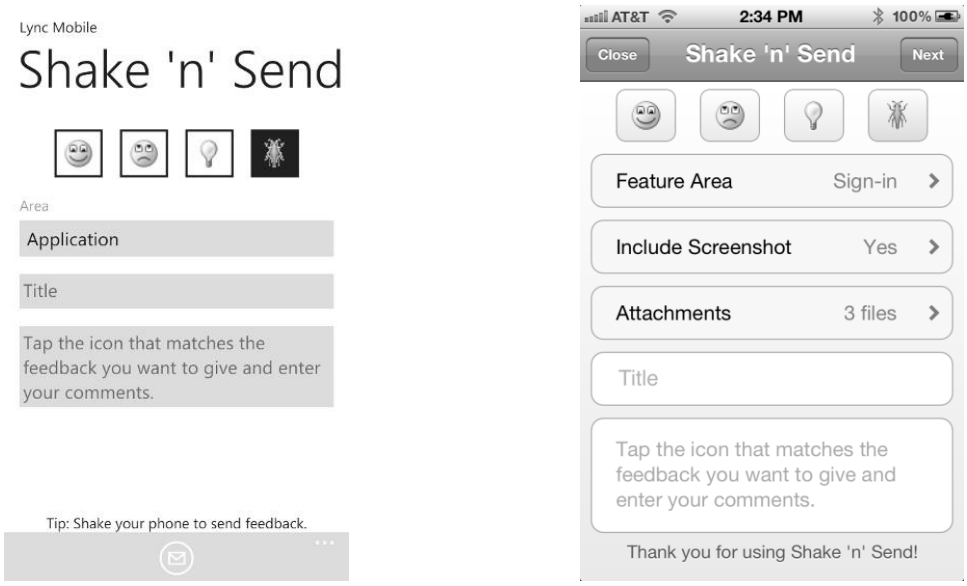


Figure 2 - Screenshots of "Shake 'n' Send" for Windows Phone 7 (left) and iPhone (right)

3.2 Using email to send feedback

The next step we took in building *Shake 'n' Send* was to determine how to send the user feedback to the product team. Since mobile applications must accommodate for unreliable network connectivity, methods are required for caching the feedback data and retrying the upload until it can be successfully sent. Depending on the size of the upload, the mobile device may take a significant amount of time and resources to successfully upload the feedback data, which may also be interrupted if the mobile user becomes impatient and terminates the application prematurely. This would require the feedback mechanism to be able to run in the background without any guarantee of sufficient resources from the operating system. Additionally, we would have to implement both the server and the client for the web service which, depending on the client platform, may not be a trivial process.

After looking at all of the challenges that a web service imposed, we took another look at the existing infrastructure of the mobile platform. We discovered that every major mobile operating system supported a native email client and that certain APIs were exposed to third party applications that could be used for sending email directly from the application. Hence, we decided to experiment with email as the transport mechanism for sending feedback. In doing so, we were able to save significant development costs by utilizing the existing email client on the device with very little additional code. The email infrastructure on the mobile platform also automatically handled all of the retry logic and background processing that a web service would have required us to implement. By using email, we could guarantee 100% reliability on delivery of the feedback data, a feat which would require significant investment by using a web service.

The only potential drawback with using email to send feedback is that it requires the mobile user to have set up an email account on the device. However, since most users use mobile devices for email purposes already, we felt that this was not a significant problem, and this has proven to be an accurate assumption in practice.

3.3 Backend service

For some product teams, simply receiving the feedback email may be sufficient. For a small amount of feedback, the team may be able to afford to manually process the data. But if the amount of feedback

increases, this manual process may become overwhelming. Hence, we decided to develop a backend service that automatically processed the feedback emails as they arrived.

First, in order to automatically process the feedback, all of the emails had to be sent to the same email account, so we created a new Microsoft Exchange email account and configured our mobile client applications to send the emails to this account. Next, we created an Exchange rule that would automatically filter the incoming emails into different Exchange folders based on the email's subject. This allowed us to filter the emails based on the client platform type (e.g. Windows Phone, iPhone, iPad, etc.) as well as to eliminate any additional noise or junk emails that may have been sent to the email account.

Once the email account was properly configured, we created a service using Microsoft's .NET framework and Exchange Web Service subscriptions to connect to the individual Exchange folders and to determine when a new feedback email arrives. Then, using the data that was sent within the email by the user, the backend service was able to act upon that data. If the feedback that was sent was marked as a bug, the backend service automatically filed a bug into our product team's bug database. After the bug was filed, a bug summary email was sent to the original sender as well as to each of the product team members to notify them of the new bug. Additionally, the feedback data was stored into a SQL database for easier reporting and analysis. Other feedback data such as positive comments and feature requests were also stored in the SQL database. Figure 3 below shows the entire end-to-end *Shake 'n' Send* process.

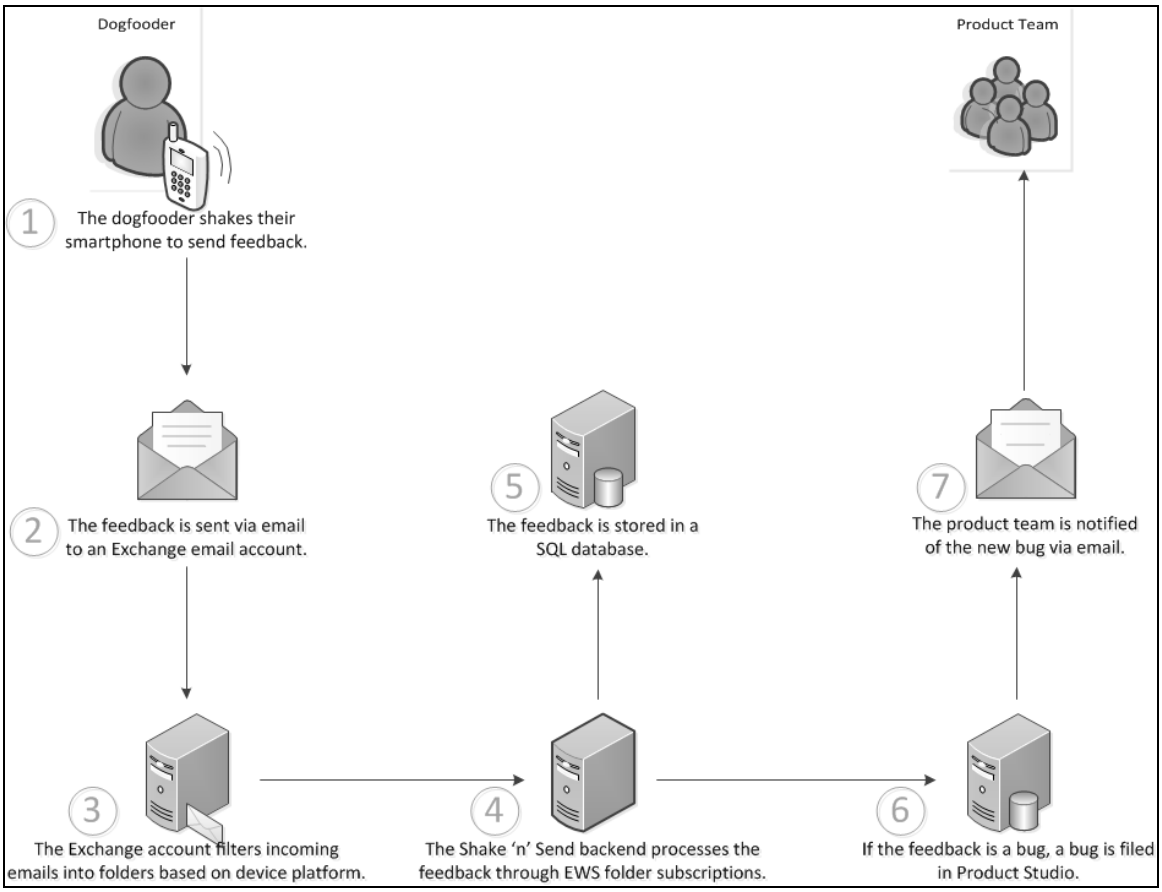


Figure 3 - The end-to-end "Shake 'n' Send" process

3.4 Additional features and benefits

Shake 'n' Send provides many benefits for sending feedback directly from mobile applications:

- **Automatic screenshot capture** – When the user shakes the device, a screenshot can be captured instantaneously, and the user can be given the option to include the screenshot with their feedback. If the user is submitting a bug, the product team can immediately use the screenshot to help understand the issue, thus saving time for both the end user and the product team.
- **Feature area detection** – The current state of the application can be detected at the time of the shake, and the feature area for which the user is sending feedback can be predicted and pre-populated in the *Shake 'n' Send* user interface, again saving time for the end user. Then, when the feedback is sent, the correct feature area owners can be notified directly instead of the entire product team.
- **Easy to integrate/remove** – Traditionally, the "Send Feedback" button approach requires tight integration between the feedback and product code. Placing a button on every view is very costly in both integration (and removal at time of release) and real estate on the small device screen. By using the shake gesture, there is no need for button real estate, and the *Shake 'n' Send* code is almost completely separated from the product code. This allows for much easier integration and removal of the *Shake 'n' Send* code within the product.
- **Familiarity** – *Shake 'n' Send* can be used by many different product teams for any mobile application. If a user has previously used *Shake 'n' Send* while dogfooding one of Microsoft's mobile applications, they will already by familiar with the feedback process and will be much more inclined to use it.
- **Fun to use** – The shake gesture is fun to use! This encourages users to send more feedback which product teams can use to improve the quality of their mobile application.

4 Results

During the beta timeframe of Microsoft Lync 2010 for Windows Phone, iPhone, and iPad product cycles, the Microsoft Lync test team deployed *Shake 'n' Send* to its internal customers to obtain feedback about the mobile applications under development. As a result of using *Shake 'n' Send*, non-product team members (other internal Microsoft users) submitted several thousand feedback reports of which over 10% were found to be unique and legitimate product code bugs that were eventually fixed before the products were released.

Unfortunately, we do not have any previous dogfooding data for mobile applications before the introduction of *Shake 'n' Send*, so we cannot analyze the impact that *Shake 'n' Send* has had on the overall feedback from dogfooders.

With several thousand feedback reports, we see that *Shake 'n' Send* was an effective way for dogfooders to send feedback to the product team. With this vast amount of feedback, the product team was able to utilize it to improve the quality of the software. Without *Shake 'n' Send*, we estimate that up to several hundred defects would not have been discovered prior to the customer release which could have drastically affected software quality. Instead, the product team was able to fix an additional 30% more bugs in the product before it shipped.

Shake 'n' Send has received a lot of positive feedback across a variety of different teams. The product team has commented that the feedback received by dogfooders has been extremely helpful in understanding the end user's perspective on the product, and this has helped the product team spend their resources to address the top issues that have been reported by dogfooders. Dogfooders themselves have also commented that they enjoyed how fun and easy *Shake 'n' Send* was to use. Because of this, many dogfooders were more inclined to send feedback more often which increased the overall amount of feedback that the product team was able to use to improve the quality of the product. The impact and effectiveness of *Shake 'n' Send* was also seen when the product team removed the tool before the product was released because the overall amount of feedback that the product team received from dogfooders dropped off significantly.

In the past year, *Shake 'n' Send* has continued to gain popularity, and other product teams at Microsoft have begun to use the tool internally for receiving feedback from dogfooders on their mobile applications.

5 Conclusions

As we have seen, gathering user feedback on mobile applications offered quite a challenge due to the nature of the mobile platform. By analyzing the benefits of mobile development as well as its drawbacks, we were able to develop a successful tool that enabled dogfooders to send feedback directly to the product team from within the mobile application. With the help of *Shake 'n' Send*, the Microsoft Lync product team was able to gather much more feedback on its mobile products than it could have otherwise. With the amount of feedback obtained, combined with the automated process of collecting, submitting and organizing defects, we were able to make significant improvements in product quality by fixing an additional 30% more bugs in the product. As the tool gets embraced by other Microsoft mobile product teams, we look forward to analyzing *Shake 'n' Send*'s abilities to facilitate improving software quality.

References

Wikipedia. "Eating your own dog food," http://en.wikipedia.org/wiki/Dogfooding (accessed August 2012)
Wikipedia. "Sandbox (computer security)," http://en.wikipedia.org/wiki/Sandbox_(computer_security) (accessed August 2012)
Microsoft. "Lync 2010," http://office.microsoft.com/en-us/lync (accessed August 2012)

Engineering Quality in the Robotic World

Vicki Niu, Ethan Takla, Ida Chow, MacLean Freed, Jeffery Wang and Kingsum Chow
{nanites4092}@gmail.com, {kingsum.chow}@intel.com

Abstract

This paper explores the challenges of attaining engineering quality in robotics through the experiences of the Lincoln High School robotics team, the Nanites.

In robotics, engineering quality in hardware-software co-design is important. While software is written in the abstract world, it is executed on robot's hardware, which interacts with the physical world to accomplish its missions. The quality of the interactions between the robot's sensors, motors, and the physical world is critical to the success of the missions.

Testing software quality on a robot is different from testing software on an information system. In the physical world, the state of a robot is never precisely known. Sensor readings inevitably contain errors and often provide insufficient information. The robot's motions, no matter how well controlled, are inexact. As a result, accomplishing a task using a robot requires tolerance of some error. Testing software quality on a robot in the physical world thus requires a different mindset than testing software quality in the information technology world.

Through the case study of building and testing a robot with motors and sensors, this paper characterizes the uncontrollable and variable environmental factors encountered in the physical world and the uncertainties from sensor readings, especially those regarding the location of the robot.

The contributions of this paper are:
1. To share the lessons learned from striving for engineering quality in robotics and hardware-software co-design.
2. To demonstrate that high school students are inspired to learn advanced sciences and mathematics not simply for exams, but for robotics competitions, and the sake of learning.

Biography

The authors are members of the Lincoln High School robotics team "The Nanites". They presented a paper at PNSQC two years ago as a FIRST LEGO League team. In 2010, they received the first place programming award in the FIRST LEGO League World Festival in Atlanta, Georgia. The Nanites became a FIRST Tech Challenge robotics team in 2010, upon entering high school. In addition to striving for hardware and software excellence, the Nanites are very involved in educating youth in the Portland metropolitan area about robotics through camps, workshops, and demos. More information about the team is available at http://nanites.zymichost.com

Kingsum Chow is a Principal Engineer at Intel. He works in the System Software Division (SSD) of the Software and Services Group (SSG). He joined Intel in 1996 after receiving his Ph.D. in Computer Science and Engineering from the University of Washington. Since then, he has been working on performance, modeling and analysis of software applications. Currently, he leads the Java Middleware performance optimization on Intel architecture. He holds more than 10 patents and he has presented more than 40 technical papers. In his spare time, he volunteers as a coach for multiple robotics teams, bringing the joy of learning about "Science, Technology, Engineering and Mathematics" to the students in his community.

1 Introduction

This paper explores how a high school robotics team, the Nanites, overcame the obstacles in attaining engineering quality in robotics. The Nanites are a FIRST TECH Challenge robotics team. FIRST TECH Challenge (FTC) is a high school robotics competition that is becoming increasingly popular.

The Nanites' work on quality assurance through robot competition dates back to 2005 when they first competed in FIRST LEGO League. Other FIRST programs include Jr. FLL and FIRST Robotics Challenge (FRC). Jr. FLL, for kids ages 6-9, consists of a challenge similar to the theme of FLL for that year. Teams create a LEGO model based on research that they do relating to the year's theme. In FRC, teams of high school students build larger robots and compete on a much larger field than in FTC.

1.1 FIRST LEGO League

First Lego League (FLL) is a robotics program for students ranging from elementary school to middle school. The students utilize a LEGO NXT brick, LEGO Mindstorms pieces, and NXT-G programming software to program an autonomous robot to complete various missions on a FLL table for two minutes and thirty seconds. Figure 1 shows the FLL field setup for the 2011-12 season. Each team builds a robot to accomplish missions while navigating around obstacles on the table. This experience introduces participants to critical thinking, team building, presenting ideas, and applying math and science to solve real world problems. FLL team members learn skills that will benefit them for the rest of their lives.

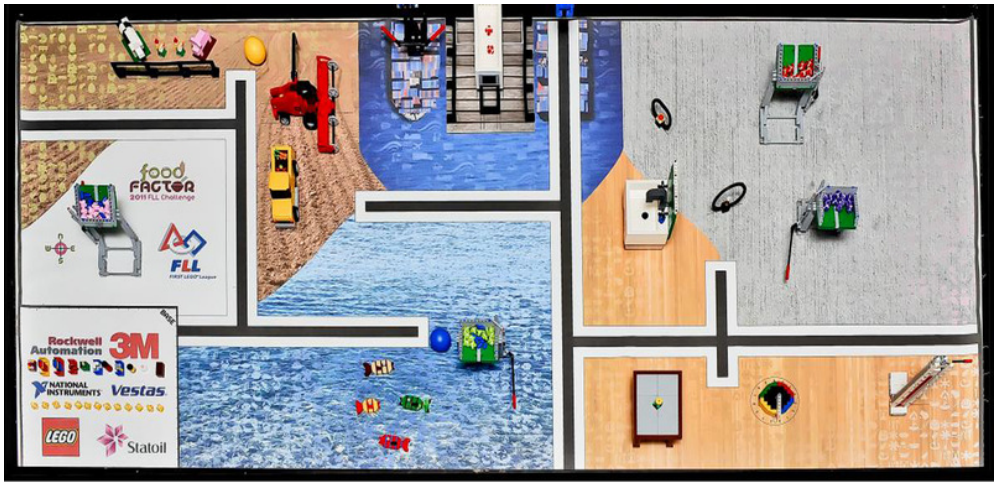


Figure 1: FLL Mat

The first years in FLL were predominately a learning experience for the team as they had no prior experience with engineering methodology. Over the years, the Nanites learned about programming, teamwork, sensors, and many important life skills. The team attempted to achieve quality assurance in progressively more complex tasks, from having an effective way to connect motors to the robot to having the most effective and reliable line-following algorithm. Such experiences led the team to discover that a group effort is better than an individual one.

1.2 FIRST TECH Challenge

In 2010, the Nanites progressed into the FIRST TECH Challenge program. There are many important differences between the FLL competitions, for elementary and middle school students, and the FTC competitions, for high school students.

In FLL, the competition field consists of a tabletop field measuring 8' x 4' [1] while in FTC, the competition field is 12' x 12' [2]. On the FTC field, four 18" x 18" robots compete against each other and must be able to navigate around each other without breaking when hit by another robot. Conversely, a LEGO NXT robot does not require the same amount of durability as the FTC robots do because it has the field to itself, leaving room for more unwieldy contraptions on the robot. This is an added complexity of FTC, as the variable of three other robots' actions are introduced, while an FLL robot is always the sole player on its field. Figure shows the field setup for the 2011-12 FTC season. Each alliance team of 2 robots moves from their bases, the squares in the lower right and upper left corners, to the field and performance various actions such as picking up racquetballs and putting them into the crates.

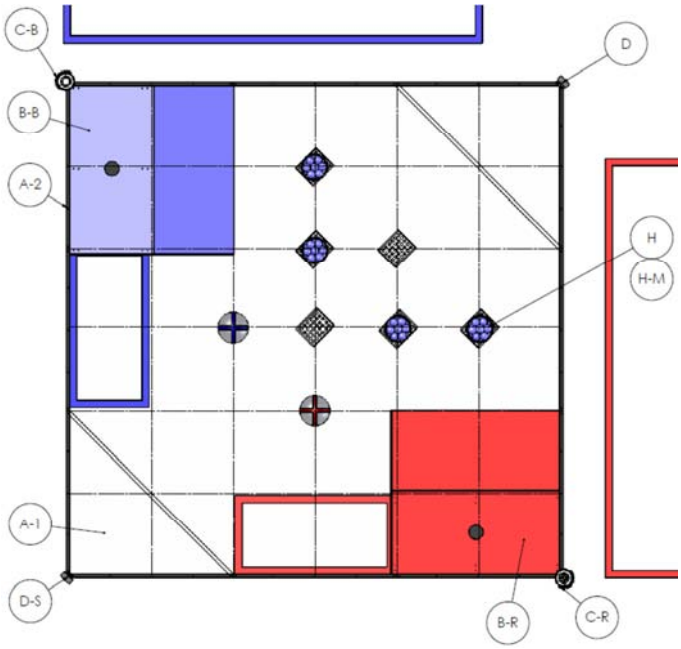


Figure 1: FTC Field Setup for the 2011-12 season

In FLL, the robots' actions are completely automated. In FTC, there are 30 seconds of autonomous robot action followed by 120 seconds of tele-operated (TeleOp) control, when two team members drive the robot with Logitech controllers to complete other goals on the field [3].

There are limited LEGO sensors available for use in FLL, such as a touch sensor, a sonar sensor, and a light sensor. In FTC, a prototype board may be used to utilize any number of specialized sensors, giving teams a virtually unlimited range of sensors that may be used [3 §4.2]. FTC teams can thus use custom sensors that allow more freedom and better results than the prescribed LEGO sensors.

Many FLL teams use NXT-G, a graphic-based programming language that has restricted capabilities. In FTC, a C-based system, RobotC, is used most widely. RobotC has data logging [5] and computational capabilities far more advanced than those of NXT-G, making it better equipped to analyze many sensor readings. RobotC also includes the Virtual Worlds package, where programs can be tested before being put on an NXT block and used with a physical robot, increasing the quality assurance of programs put on the robot.

1.3 Sensor Reliability

FTC is a good case study of software quality. Although consumer-level sensors have become more accurate in recent years, most sensors are still erroneous to some degree. These errors, if uncorrected, can result in unreliable and low-scoring robot performance. The quality of software written to compensate

for these errors can be tested in the physical world using qualitative observations and/or a robot localization system. Many sensors included with the FIRST TECH platform show much inaccuracy; to attain reliable performance, the students may choose to develop their own, more accurate, custom sensors.

Through the case study of building and testing a robot with motors and sensors in FTC, this paper characterizes the variable environmental factors encountered in the physical world, many of which are uncontrollable. The paper characterizes the uncertainties from sensor readings, proposes solutions to address these uncertainties, and then evaluates the effectiveness of those solutions.

2 Software Quality Issue: Achieving a Known State

The autonomously controlled period in FTC requires that the robot know its location. As stated in Cox 1991, "using sensory information to locate the robot in its environment is the most fundamental problem to providing a mobile robot with autonomous capabilities." In competition, there is infinite variability and complexity, so the robot is incapable of ever truly knowing its position.

Determining the robot's location is difficult because the factors in the physical world are constantly changing. Recognizing this, the authors used sensory technology in an attempt to gain information with regards to the robot's environment. Although this did increase the robot's certainty of its own position to some degree, many new complications arose due to the unreliable nature of the sensors. The commonly-used sensors in FTC are problematic. A LEGO-provided ultrasonic sensor, which uses ultrasonic waves to measure the distance from a target, was tested for its accuracy and reliability. As seen in the figure below, the readings have large amounts of error, making for unreliable performance

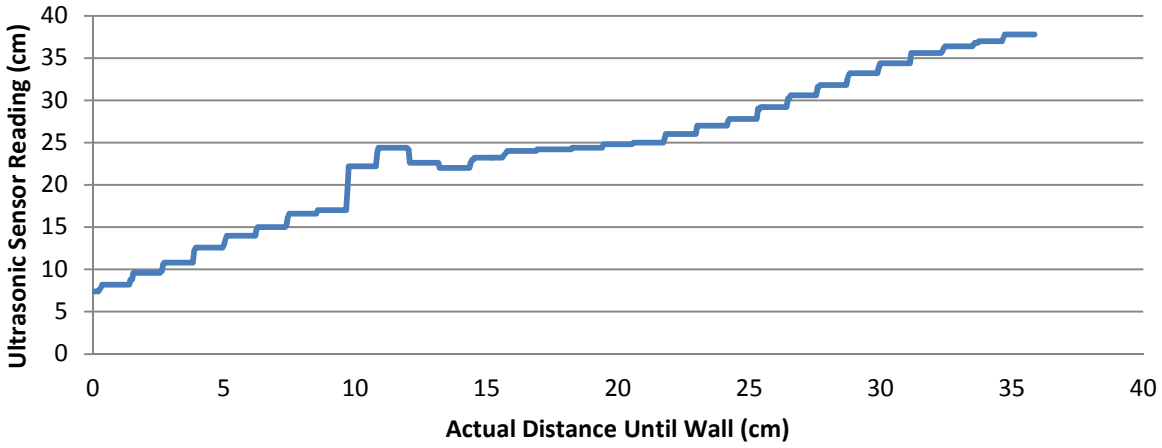


Figure 2: Ultrasonic Sensor Reading Error

Other sensors used were quite reliable, such as the LEGO light sensor, but simply insufficient. The robot's tasks required different types of information than those provided by the sensors used. In a playing field largely the same color, light sensors contributed little to the robot's knowledge of its location. The robot needed information to let it know, in as direct a manner as possible, of its orientation. The team attempted to use HiTechnic-provided compass sensors which take information with regards to the strength of the earth's magnetic field along three axes and output a heading, but this heading proved to be inaccurate and not tilt-compensated. A magnetometer, which returns the strengths of the three axes of the earth's magnetic field in milligauss, not a heading, was also tested, but its values were greatly altered by both hard and soft iron distortion.

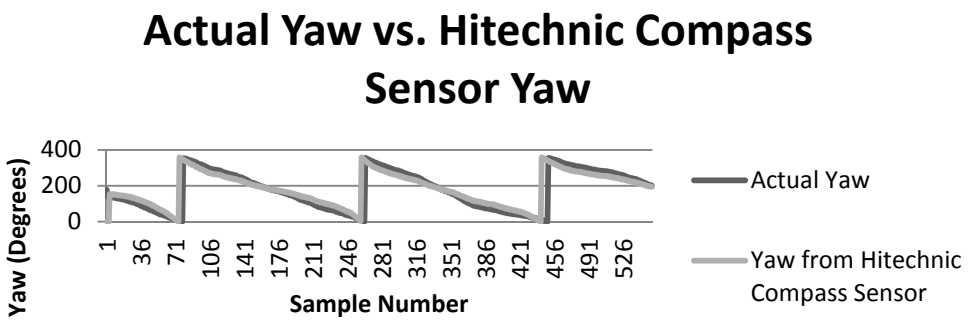


Figure 3: Graph of HiTechnic Compass Sensor Yaw Error

HiTechnic accelerometers, which measure the acceleration of the robot along three axes, were also evaluated in an attempt to get accurate roll, pitch, and yaw values. While the sensor was accurate while stationary, it exhibited errors of up to 50 degrees while moving, rendering it essentially useless. With the HiTechnic gyroscopes, which output a rotation rate in degrees around each of the three axes, there was found to be a slight drift over time that was difficult to compensate for as the offset was constantly changing. Although these sensors provided the correct information, they did so unreliably. The ultimate complication in achieving a known state is finding a precise, accurate sensor that provides relevant information.

3 Solutions to Software Quality Issues

3.1 Attitude and Heading Reference System

The magnetometer error was mitigated by taking the magnetometer values in a variety of orientations, and then calibrating the sensor to correct for the hard and soft iron distortion.
Although it was found that the accelerometer and gyroscope alone cannot give accurate attitude values, it is possible to use both sensors to achieve the desired results. This can be done by using a direction cosine matrix (DCM), a computationally efficient attitude estimation algorithm [10]. It integrates the gyro values in order to obtain an angle, but also detects any drift by the gyro and compensates for it using a PI feedback loop.

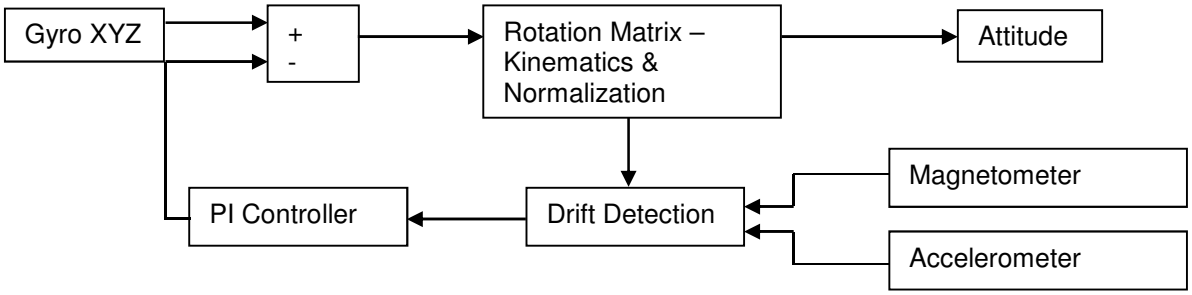


Figure 4: PI Feedback Loop

This consolidated system is called the attitude and heading reference system, or AHRS. It fuses 3-axis accelerometer, 3-axis gyroscope, and 3-axis magnetometer values to obtain readings of the robot's roll, pitch, and yaw.

3.2 Image Tracking Using Cameras

To localize itself, the robot may also be equipped with two cameras that feed visual data to an onboard processor which identifies landmarks and calculates the distances to them using a semi-global block matching (SGBM) algorithm.
Most low cost cameras today do not have perfect lenses, and do not have the same internal mounting, meaning that two camera images can have 2D offsets and distortion. In order for the SGBM algorithm to work, the two cameras need to be rectified so that their image outputs are aligned. The calibration was done using a chessboard, as the corners are easily recognizable. The algorithm first identifies the corners on the chess board, then uses those points to calculate and correct for distortion and offset.

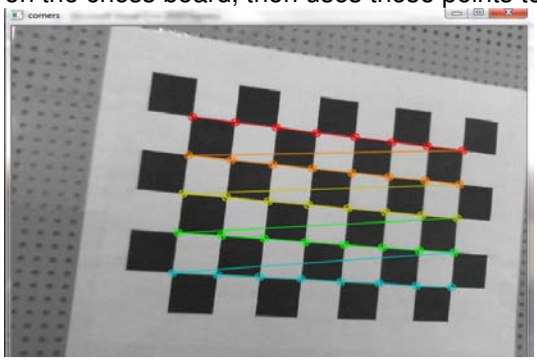


Figure 5: Chessboard Calibration of Camera

Using rectified image pairs from the left and right cameras, the SGBM algorithm finds a cluster of pixels from the right camera image and matches it to the most similar section of the left camera image. The SGBM algorithm then calculates the pixel disparity for those clusters and calculates the distance from the camera using the parallax effect; a closer object will have greater pixel disparity than a further object. Below is a pair of rectified camera images, and their corresponding disparity map.

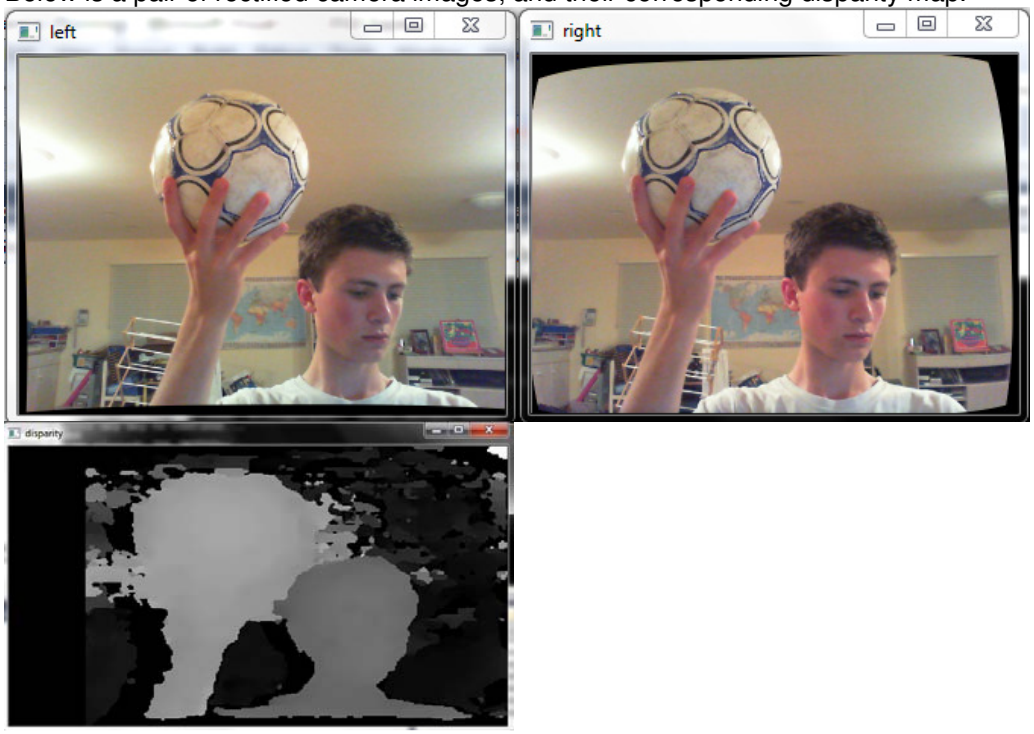


Figure 6: Calculated Disparity Map from Above Images

In order to calculate the distance from an object using the disparity map from the SGBM algorithm, one must first measure the distance between the two camera lenses. An object is then placed in front of the two cameras at a known distance, and the SGBM algorithm is run. The resulting disparity of the object is then used to calculate a calibration parameter that enables distance calculation.

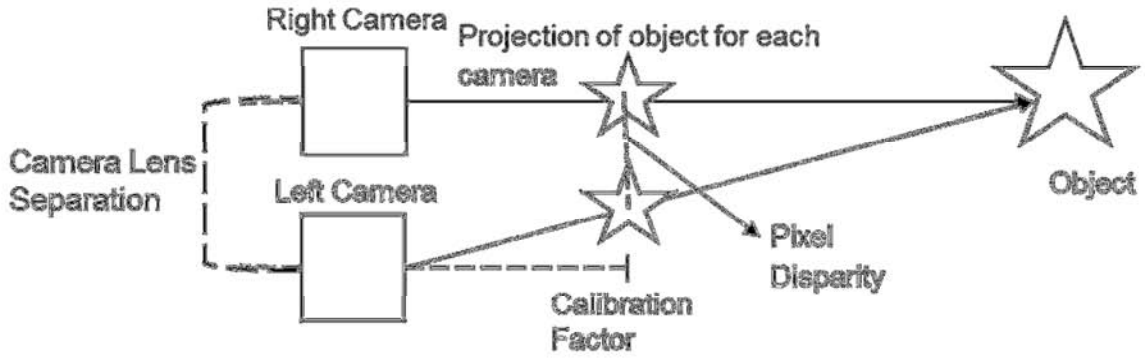


Figure 7: Distance Calibration Setup

4 Evaluating Efficiency of Solutions

4.1 Efficiency of Achieving a Known State Solution

The attitude and heading reference system was proven to be much more reliable than the HiTechnic Compass sensor. Due to the calibration, the AHRS returns an accurate, tilt-compensated heading. This can be seen in the graph below.

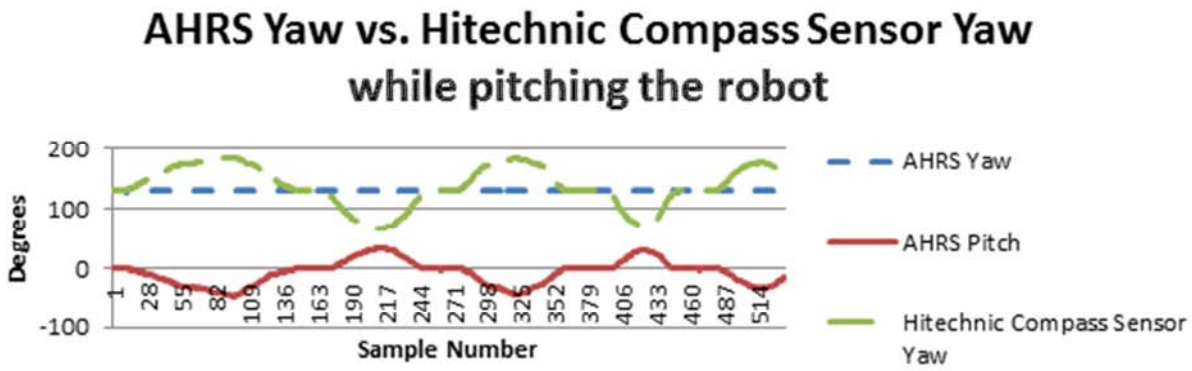


Figure 8: AHRS vs. HiTechnic Sensor Yaw

With accurate information with regards to its roll, pitch, and yaw, the robot is much more able to determine its position and orientation in the field. Without interference from hard or soft iron distortion, or misrepresented values due to acceleration, the process of truly knowing where our robot is position is much better facilitated.

5 Future Plans

With respect to the image-tracking algorithm used, there is still much work to be done. The team has yet to implement this solution on the robot and test it in real-time in a competition scenario. There will be much work to do in improving the efficiency of the system and fine-tuning the algorithm. The team plans to continue testing with this idea so that the efficiency of using cameras to localize the robot can be evaluated and further progress can be made.

While these endeavors with achieving higher levels of software quality have been fruitful, there is yet much work to be done. The efficiency of the proposed solutions has been evaluated, and there is much room for growth. Furthermore, there are many avenues leading to the known state and efficient movement which have not been discovered, and thus the team plans to explore these in times to come.

Acknowledgments

The authors wish to thank the FIRST Tech Challenge community, especially the Oregon Robotics Tournament and Outreach Program, for providing such a wonderful opportunity for high school students to learn about technology. The authors would also like to think Mr. Richard Vireday and Mr. Aaron Akzin for their invaluable suggestions to improve this manuscript.

References

[1] FIRST LEGO League, “Food Factor Challenge,” [Online], Available: http://firstlegoleague.org/sites/default/files/Challenge/FoodFactor/FLL2011_Complete_Challenge.pdf

[2] FIRST Tech Challenge, “Bowled Over! Challenge,” [Online], Available: http://www.usfirst.org/sites/default/files/uploadedFiles/Robotics_Programs/FTC/Game_Info/2011/Complete-Build-Guide.pdf

[3] FIRST Tech Challenge, “Bowled Over! Game Manual,” [Online], Available: http://www.usfirst.org/sites/default/files/uploadedFiles/Robotics_Programs/FTC/Game_Info/2011/Bowled-Over-Game-Manual_Rev%285%29.pdf

[4] RobotC, “RobotC NXT Curriculum,” [Online], Available: http://www.robotc.net/education/curriculum/nxt/

[5] V. M. B. K. Pirabakaran, “PID Autotuning Using Neural Networks and Model Reference Adaptive Control,” *Proceedings of the 15th IFAC World Congress, 2002*, 2002. [Online]. Available: http://www.nt.ntnu.no/users/skoge/prost/proceedings/ifac2002/data/content/01467/1467.pdf. [Accessed: 15-Jun-2012].

[6] W.-yong Han, J.-wook Han, and C.-goo Lee, “Development of a Self-tuning PID Controller based on Neural Network for Nonlinear Systems,” *Control*, pp. 979-988, 1999.

[7] F. Lin, R. D. Brandt, and G. Saikalis, “Self-tuning of PID controllers by adaptive interaction,” *Proceedings Of The American Control Conference*, vol. 5, no. June. American Autom. Control Council, pp. 3676-3681, 2000.

[8] D. Brutzman, “From virtual world to reality: designing an autonomous underwater robot,” 1992.

[9] Y. Kuroda, K. Aramaki, and T. Ura, “AUV test using real/virtual synthetic world,” *Proceedings of Symposium on Autonomous Underwater Vehicle Technology*, pp. 365-372.

[10] S. O. H. Madgwick, “An efficient orientation filter for inertial and inertial / magnetic sensor arrays,” Report xio and University of Bristol UK, vol. 2011, p. 1--32, 2010.

[11] Oliver J. Woodman, “An introduction to inertial navigation,” University of Cambridge, p. 1--37, 2007.

An Action Model for Risk and Uncertainty in Decision Making - How to Avoid the Paralysis of Analysis

Jeffrey A. Robinson, Ph.D.
jeff.robinson@aqionline.com

Abstract

It is fairly well accepted that 'risk' and 'uncertainty' impact decision making activities and make them more complex and difficult. Unfortunately, it is not always clear, from the way we use these terms, how risk and uncertainty influence how we should act when complex decisions are being considered. Part of the problem is that there are several different terms used to refer to these complicating factors. The words 'risk', 'impact' and 'cost' are used almost interchangeably… as are 'probability', 'frequency', 'likelihood', 'certainty', or 'uncertainty'. To a great degree, when any of these factors are present we tend to analyze problems more carefully in order to make the best choices and to avoid potentially undesirable outcomes. There are many different analytical techniques that can used to break down details of a problem, explore the underlying factors to project likely outcomes and prioritize possible choices and courses of action. These include Failure Mode Effects Analysis, regression analysis, correlation analysis, root cause analysis, Ishikawa diagrams, Cause-Effect matrices, and a myriad of quantitative risk analysis techniques. However, this general tendency to analyze things when risk is perceived can be detrimental and can lead to a phenomenon known as the "paralysis of analysis" in which far more time is spent perusing and processing data than is necessary or appropriate. To understand how risk or uncertainty affect the decision making process, we must first carefully define these terms and examine the nature of choice and action taking with a more formal approach. This paper presents two important concepts. The first is a rigorous definition of risk and uncertainty. The second is a taxonomy that clearly suggested different modalities of action for the different combinations of these two factors. Hypothetical case studies are provided to illustrate the rationale behind this action model, which clarifies when quantitative analysis 'is' and 'is not' appropriate. This model can be applied to projects of all types including software development and process improvement projects. If used during project selection and prioritization, it can be effective in avoiding excessive analysis activities and minimize the tendency to overanalyze whenever risk or uncertainty are present in any proportion. Developing a formal methodology for assessing and managing risk is important to optimize utilization of resources and mitigate the effects of potentially undesirable outcomes. Knowing when and how to analyze risk is important, but it is equally important to understand when such analysis is not warranted and would only cause unnecessary cost and delay. Managers and decision makers need to understand that decisive action without excessive risk analysis is still possible when risk and uncertainty occur alone. Only then will they be able to act confidently and appropriately when either or both are present.

Biography

Dr. Jeffrey Robinson is the Vice President, Principal Consultant, and Co-founder of Accelerated Quality Improvement, a leader in process improvement consulting and training. He is the co-author of Six Sigma Software Quality Improvement – Success Stories from the High-Tech Industry, McGraw Hill, 2011. Dr. Robinson is a senior technologist and a Six Sigma consultant with more than 25 years experience in IT and manufacturing with companies such as Motorola, Medtronics, and Rockwell. He has served as Master Black Belt, Division Quality Manager, Manager of CIM (Computer Integrated Manufacturing), and System Manager of large automated factories. He is a frequent author and lecturer and has taught many Six Sigma courses.

1 Introduction

It is fairly well accepted that 'risk' and 'uncertainty' impact decision making activities and make them more complex and difficult. (Riabackel, 2006). Unfortunately, because of the way we use these terms, it is not always clear how we should act when complex decisions are being considered. (Abdellaoui and Hey 2008)

Part of the problem with the way that risk and uncertainty affect decision making is that there are several different terms used to refer to these complicating factors. The words 'risk', 'impact', and 'cost' are used almost interchangeably… as are 'probability', 'frequency', 'likelihood', 'certainty', or 'uncertainty'.

To a great degree, when either of these factors are present in any measure, we tend to analyze problems more carefully to help us make the best choices and to avoid potentially undesirable outcomes. The discussions that follow focus on the choices and actions that people make when there is uncertainty about what to do or how to act. Sometimes, what people decide to do is to conduct further analysis, to help them make a more accurate choice or selection. Sometimes people act directly, choosing a specific alternative that they have identified or a solution or action plan. The dilemma that decision makers face lies in how risk and uncertainty combine to make this selection of appropriate actions more difficult.

There are many different analytical techniques that can used to break down details of a problem, explore the underlying factors to project likely outcomes, and prioritize possible choices and courses of action. These include Failure Mode Effects Analysis, regression analysis, correlation analysis, root cause analysis, Ishikawa diagrams, Cause-Effect matrices, and a myriad of quantitative risk analysis techniques. However, this general tendency to analyze things can be detrimental and can lead to a phenomenon known as the "paralysis of analysis" in which more time is spent studying and processing data than is necessary or appropriate. (Bensoussan, Fleisher, 2008)

To understand how risk or uncertainty affect the decision making process and to determine when extensive analysis is appropriate (and when it is not), we must first carefully define these terms and examine the nature of choice and action taking with a more formal approach. (Harrison 1998 and Sewell 2009)

2 Definitions

First, let's examine the differences between Risk and Uncertainty.

Risk maps most closely to COST, that is the cost or impact of a particular problem or alternative. Risk is the magnitude of loss (in time, money, effort or lives) associated with a specific choice or decision. It is how much might be lost (or gained) based upon different actions or alternatives, which might result from a decision that is made, or not made. (It is important to remember that taking no action and not selecting an alternative is always a valid alternative in any problem or decision-making process and may indeed be one of the most costly alternatives in the long run.)

Sometimes risk can be quantified quite precisely. At other times risk is probabilistic in nature. Risk is a factor that linearly corresponds to cost. Sometimes risk includes considerations of likelihood, or even detectability, but in all cases risk is proportional to cost. That is, if for a given likelihood or detectability, if the cost of a particular failure or a course of action is twice as much as another, then the risk is also doubled. Thus risk relates most closely to the impact or cost of a particular choice, alternative, action, or outcome (see Figure 1). When other factors are not clear, it is easiest to equate risk with potential dollar loss. When probability is considered, risk is the probabilistic cost. That is, how much money might one lose if a particular event occurs or a particular choice is made. Basically, the greater the potential financial impact, the greater the risk.

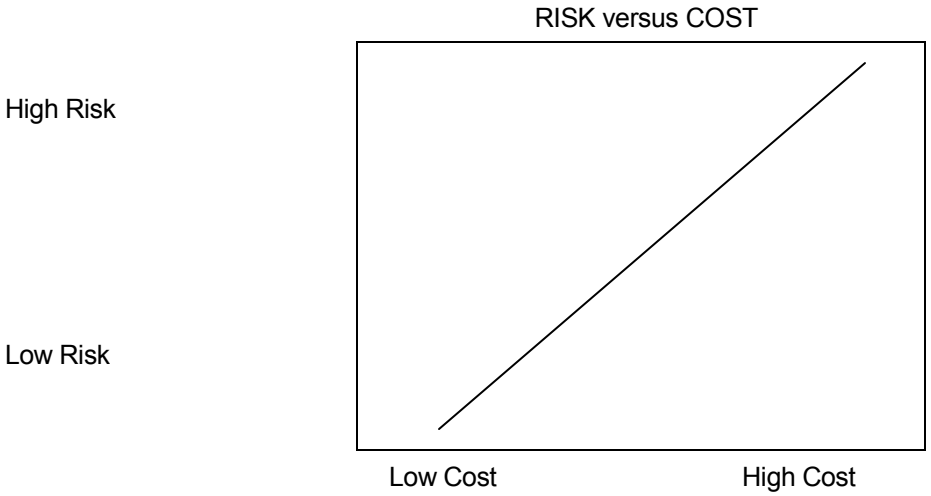


Figure 1 – Risk and Cost

Probability, on the other hand is the likelihood of a given outcome associated with different events, actions or choices. Probability is the degree to which the associated risk (or quantified loss) is likely to occur. It is usually expressed as a number ranging from zero to one. Zero means there is no chance of an event occurring. One is absolute certainty that it will occur. It may also be represented as a percentage from zero to one hundred.

It is important to note, however, that Probability is NOT the same as *Uncertainty*. This is an important distinction, since uncertainty and not risk is a primary drive in selecting appropriate actions and alternatives. Unfortunately probability is too often confused with uncertainty and, while they are related, they are quite distinct from one another. Contrary to what one might initially think, high uncertainty does not occur when probabilities are low.

Indeed, when probabilities are *very high* or *very low* (nearly one or almost zero), there is almost absolute certainty. That is, when the probability of an event is very high, then that thing is almost certain to occur. Conversely when the probability of an event is very low, the event is almost certain to not occur. The greatest uncertainty lies in the range between these two extremes. *Uncertainty* is at its maximum in the middle of this region, when the probability of an event is 50%. (Figure 2)

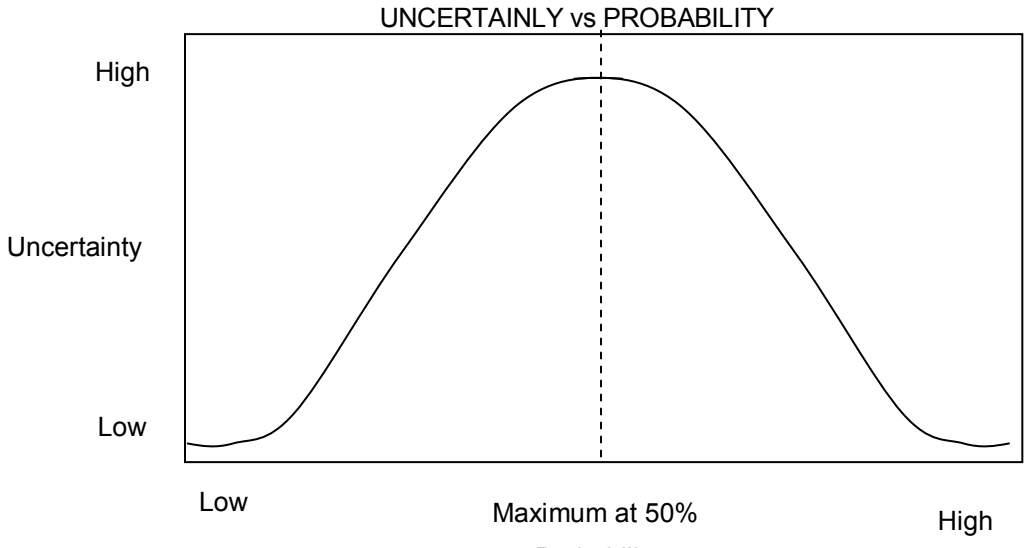


Figure 2 – Uncertainty and Probability

If we consider the situation where probability is very high or low, uncertainty is minimized. Events associated with these extremes are either very certain to occur or to not occur. Uncertainty in these regions is thus very low. It is in the middle that uncertainty is the greatest; when the outcome is as likely to occur as not. (Remember then that uncertainty is not the same as probability.)

3 A Taxonomy of Uncertainty and Risk

Now, let's examine different situations when risk occurs in conjunction with uncertainty. Below is a graphical taxonomy of uncertainty and risk (Figure 3). In the table below, four different combinations of high and low risk and uncertainty are identified.

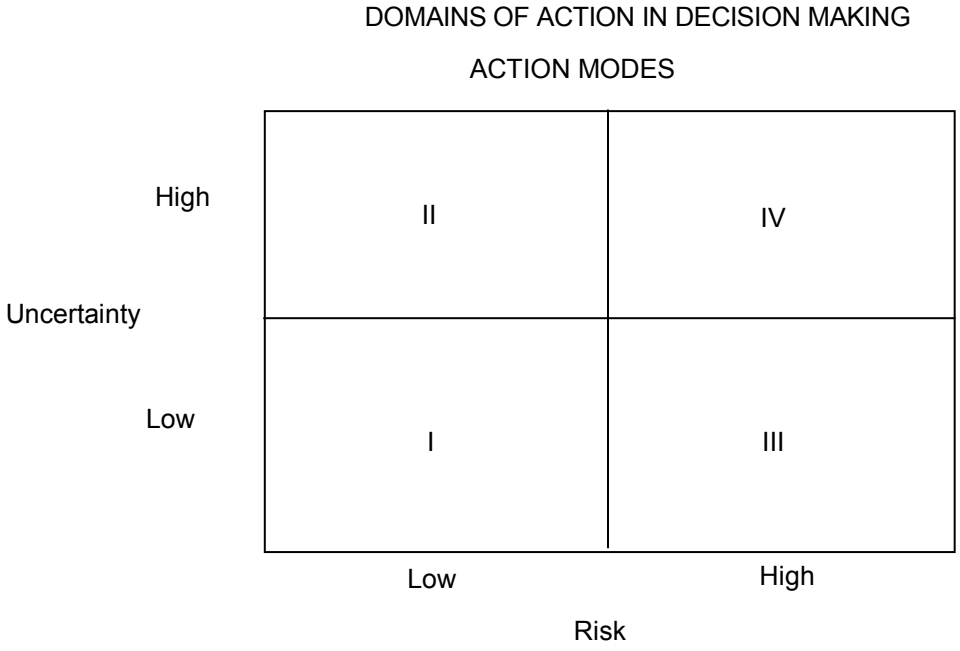


Figure3 – A Taxonomy of Risk and Uncertainty

As we examine these situations we will find that each of these domains (or regions) has a different type of optimum approach for decision-making.

For the sake of discussion each of these regions will be numbered.

Region I - Low Uncertainty/Low risk
Region II - High Uncertainty/Low risk
Region III - Low Uncertainty/High risk
Region IV - High Uncertainty/High risk

Let's examine these different domains individually.

3.1 An Analogy

3.1.1 Region I - Low Uncertainty/Low Risk

For the sake of these discussions, we will create a hypothetical problem that has low uncertainty and low risk.

Let's say that we have a problem with a large, costly software system. An examination of the system reveals a bug in the code. Moreover, the examination of the code reveals that the root cause is associated with a single subroutine and that the fix is a relatively easy one. In this case, uncertainty is low, because we know exactly what solution to implement and the effort to do so (i.e. the cost and therefore the risk) is low

Assuming that there are no additional difficulties with scheduling of performing the correction, the solution to this problem is simple.

Action: Just do it.

Our course of action is clear. Simply fix the erroneous code, test it and correct the problem. Resolving this problem does not require any special root cause analysis, quantitative risk assessment methodologies, or statistical techniques. You just implement the solution and do it. (Don't we wish all problems were this simple?)

3.1.2 Region II - High Uncertainty/Low risk

Next, let's consider a different situation. In this case we will modify our original scenario and increase the uncertainty, but not the risk.

Once again, we have a problem with a large complex software system. However, this time we find that a subroutine call to a sorting utility is malfunctioning and the sorting algorithm needs to be rewritten (Perhaps the current method only works on small files and the application will be sorting very large files instead.) In any case the current method is unacceptable and a different one will have to be applied, and there are a half dozen possible methods that can be used. Unfortunately, in this case, it is not certain which of the alternative methods will work the best. While solutions are known, it is not clear which solution will be adequate and there is no clear consensus by the programming team on which method to use. Some prefer one technique; others argue for alternatives. The problem now is that we don't know which subroutine to write to replace the inadequate one.

The good news is that method is relatively easy to write and the effort would only take a few hours to code and implement. Thus, uncertainty is high, but risk (or cost) is low.

Even if we narrow down the selection of one or two methods, we still face a problematic decision point. We need to determine which algorithm to use and we need to make sure it works adequately. So how do we decide which solution to implement? How do we pick the correct algorithm?

So what course of action do we take?

Action: Select one of the alternatives and test empirically.

That's right! When the risk of failure (i.e. the cost of an action or choice) is very low, then one need not worry overmuch about making bad choices. Indeed, making fast choices, even though they are the wrong ones, may be far better than trying to only make the best choice. This is called making a satisfying decision rather than a maximizing one.

In other words, the best course of action in this case is to simply write up code corresponding to the most favored options and alternatives and test their ability to sort large files. If the first subroutine doesn't work, try the next one instead. If that one doesn't work, try the third. If there is no risk to writing these subroutines and trying them out, then the fastest way to find a solution that works may be to systematically test different alternatives and until one is found that works. Moreover, you do not have to test all possible sort methods, because any one that is sufficient would be adequate. This leads to the rapid identification and selection of an appropriate replacement algorithm with minimal delay and (because of low risk or impact) at minimal cost, as well.

Once again detailed analysis of the different methods is not necessary. Experimentation is faster, simpler and cheaper.

3.1.3 Region III - Low Uncertainty/High Risk

Moving on to yet another situation, we will consider the scenario where we have a new high risk but low uncertainty.

In this third case, our system has again broken and needs to be fixed. This time, however, uncertainty is low. We have found that the current database used by the system no longer works. The freeware DBMS that was originally used to process SQL queries has reached its performance limit and the current number of and size of tables exceeds its capabilities. It has been determined that there are a couple other databases that could be used to replace the original DB. Specs indicate that either of the options could simply be plugged in and used with minimal impact to the system. However, this time the impact or risk is very high because the replacement DB for the system is not free and is, in fact, quite expensive. Our problem now is to decide, should we buy either of the new databases? In this particular scenario, Uncertainty is low (we know that either of the chosen solutions will work and that either will exceed current system requirements), but risk is high (since the solution is very expensive).

So what is our best course of action?

Action: Make a judgment.

When this type of problem occurs, one must determine whether or not the purchase of the solution is worth the cost. To answer that question, one typically needs to perform a simple cost/benefit calculation.

For instance, the decision maker might ask several questions, such as:

- How long would the system have to run before it pays for the cost of the new DB?
- Is there sufficient revenue or service provided to justify the upgrade?
- Would it be cheaper to buy a new system? Or can the upgrade be justified?
- Or should the current system be trimmed down to run in a degraded mode?

This is the scenario when quantitative methods can best be used to make the decision. The numbers will determine whether or not the decision to upgrade or not is the right one. There is, however, little analysis to be performed. The action of choice and selection primary consists of making a quantitative judgment. A cost benefit analysis, or the use of weighted attribute analysis could be used to show which alternative is best.

For instance, if the system is a legacy system that is rarely used, there may simply not be enough potential revenue to justify upgrading the system. On the other hand, if the system is a critical component of a core business process that facilitates financial transactions of more than a half a million dollars every day, then you want that system upgraded and fixed as quickly as possible. The choice is based upon a quantitative judgment focusing on the cost of the different options, but there is minimal analysis involved.

3.1.4 Region IV - High Uncertainty/High Risk

In our final situation, we address our worst case scenario, then risk and uncertainty are both very high.

Once again, our system is not working properly. Performance has degraded. information is being lost and the system keeps crashing. This time it is not clear which part of the system is broken or what needs to be done to fix the system. The programmers who write the system a few years ago left the company recently. No one knows where the manuals are and the system is no longer under vendor maintenance.

The cost of the fixing the system could range from fixing a few critical bugs to overhauling or even replacing the system. Thus both uncertainty and risk are high. The possible options are varied and the best solution, or even an adequate one, is not yet obvious to anyone.

So what do we do?

Action: Analyze; conduct additional research.

In this case, one cannot simply go ahead and start writing code or selecting replacements for specific subsystems. It is also not acceptable to use a trial and error method of employing patches across the system, because the cost of such experimentation could be exorbitantly high and costs of delays could exceed the cost of replacing the entire system. Similarly, we cannot conduct a cost/benefit analysis, because we don't know what part of the system needs to be fixed and therefore we cannot assess the value of a solution.

When this type of problem is faced, either uncertainty or risk must be reduced before a decision can be made. Additional study needs to be done to identify which part has broken or how much it will cost to find out and fix it.

Possible methods of analysis might include: examination of symptoms and failure modes, root cause analysis, fault signature analysis, examination of failure metrics, error reports, and implementation of special diagnostics. Several quantitative methods were noted earlier. In any case, multiple alternatives need to be identified and evaluated.

3.2 The Purpose of Analysis

It is important to note the fundamental purpose and objective of analysis. The reason we need to analyze and study the situation more is either to:

1. Reduce uncertainty (and revert to a Region III action mode) or to
2. Reduce risk (and revert to a Region II action mode).

It is not clear which of these two outcomes may result from analysis that is performed. Eventually you will understand with greater certainty the factors associated with the problem or you may discover cheaper and safer alternatives that will reduce the cost or risk of selecting a course of action. In either case, when either of these are achieved, the decision becomes easier as you transition into decision Regions II or III.

At first glance it seems as though the combination of high risk and high uncertainty actually promotes inaction (at least while some subsequent analysis is performed). However, the amount of analysis (and hence the degree of inaction) is itself limited.

No matter how much analysis you perform, you can never completely eliminate either Uncertainty or Risk (though many people try.) You could spend years mulling over different options without taking any action and you would might never completely eliminate uncertainty and find a solution that is guaranteed to succeed. Too much analysis would be devastating. Not only does the analysis cost time and money, every minute the system is down equates to lost productivity or lost revenue. There is, in fact, a maximum amount of analysis that should be performed. Eventually, you could spend more money analyzing the problem that the cost of completely replacing the system..

Indeed, one can quantitatively calculate the amount of analysis that should be performed. This technique is called "The Price of Perfect Information" and equates to the point where the cost of analysis exceeds the cost of committing to (or selecting) alternative courses of action, even if they are wrong (Bodily, 1998; Evans 2007). If you spend more money than this, by researching or analyzing the problem, then you are spending more than the risk of simply guessing or randomly selecting a solution. Unfortunately, some managers, who are risk averse and who cannot adequately differentiate risk and uncertainty, tend to analyze far too often and too long.

In the case of our model above, if the combined cost of delay and the cost of analysis exceeds the total cost of the system then it would be simpler to just buy a new system (COTS) and install it. The very act of not

deciding and performing extensive analysis may mandate a specific loss ($) and become a fiscal certainty. (In such a case, the decision reverts to a Region III decision scenario of high cost and low uncertainty.)

3.3 Action Model Summary

Let's review these modalities of suggested action one final time (See figure 4). Note: each domain has a different decision making strategy. remember, the decisions we are making are related to the selection of alternative courses of action or the selection of specific solutions. Risk relates to the cost of making (or not making) a choice. Uncertainty relates to our confidence that the solution will provide the desired outcome or remedy and reflects the degree of understanding about the problem domain.

- In Region I, you simply implement the obvious solution.
- In Region II you can perform empirical tests to find the right solution by trial and error.
- In Region III, you may use quantitative methods to make formal judgments.
- In Region IV, you need to go through a full analytical decision-making process (setting objectives and requirements, identifying and weighing alternatives, selecting alternatives, etc.).

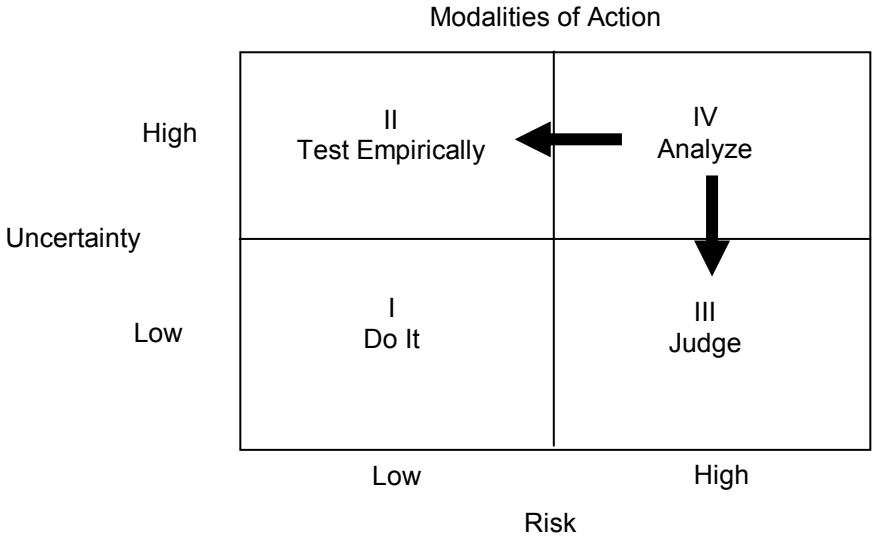


Figure 4 – Objectives of Analysis

In the case of Scenario I (low uncertainty and risk), the decision is an easy one. No extensive analysis is necessary. Similarly, it would be unnecessary to perform a full decision analysis in the case of Scenario II, since it would be faster to test alternatives and try them out than to perform analysis to infer the best choice. Experimenting empirically and testing and failing quickly can lead one to a satisficing decision with far less delay and effort.

Region III also does not warrant full analysis. However, it does justify a cost benefit calculation upon which a decision can be confidently made.

The only domain where extensive analysis is appropriate is Region IV, when both risk and uncertainty are high. Indeed, the purpose of analysis is to reduce uncertainty or risk sufficiently to drive the problem into a domain where a specific action is indicated.

Unfortunately, many managers do not adequately distinguish risk from uncertainty. To them risk and uncertainty are both equally bad and they are not clearly separated from one another. They thus confuse these different domains and different modalities of action are also blurred and indistinct.

The author has seen arguments continue for months over the selection of the best algorithms (for sorting, access speed, or efficiency) when simple experiments would have resolved the issues in a few hours.

Similarly, some companies have literally spent years performing AS-IS analysis of systems in preparation of product selection, when such analysis merely delayed a selection process for replacements when any of the identified alternative would have exceeded current system requirements and could have been completed in a few days.

When confusion between uncertainty and risk occurs, relatively simple decision methods become associated with situations that require full, detailed, analysis (like the situation described by Region IV). Sometimes, quick, easy solutions get bogged down in overly complicated decision making processes that are both unnecessary and inappropriate.

Note that extensive analysis is not always necessary even in region IV. There is great evidence that just a little analysis is often enough to lead on to the best course of action (Gladwell 2001) and extensive analysis does not improve the accuracy of our decision making, but rather merely improves our confidence in the choice or selection (Heuer 1999)

When risk and uncertainty are confused with one another, the best course of action can be obfuscated and obscured. When the distinction between risk and uncertainty is not understood, or when decision makers are unwilling to accept any amount of uncertainty or risk, then the process of making decisions becomes are arduous one. Uncertainty and risk often blur and merge, especially when managers are risk-averse or when they cannot accept either in any measure. When this is the case, the decision process bogs down with a wasteful focus on gathering and processing data to make people feel better about their choices, long after the best options may have been identified. Indeed, much of the perception that analysis is needed is driven by non-rational processes (Ariely 2008 and Harrison 1998)

In such cases, where processes of bounded rationality dominate decision making environments, managers basically become incapable of making decisions except when they are very simple ones (Region I - low risk and low uncertainty). When this situation occurs, managers perform analysis activities in regions II, III and IV (when analysis is really only warranted in Region IV where there is both high risk and high uncertainty).

As noted previously, this often leads to the ”paralysis of analysis”, where participants analyze endlessly instead of acting.

Ultimately, managers need to differentiate and distinguish risk from uncertainty in order to recognize when different types of actions are called for.

4 Conclusion

The inability to differentiate between Uncertainty and Risk can lead to unnecessary analysis and preclude appropriate actions. To avoid this, decision makers must understand how Uncertainty and Risk differ from one another and how their different combinations suggest different courses of action.

Managers and decision makers need to learn that it is impossible to eliminate all uncertainty and should focus more on appropriate modalities of action. They need to understand that action is still possible when risk and uncertainty occur alone. Only then will they be able to act confidently and appropriately when either or both are present.

References

Abdellaoui, Mohammed; Hey, John D. (Eds.); Advances in Decision Making Under Risk and Uncertainty; Series: Theory and Decision Library C, Vol. 42 2008, XIV, 242 p. 57 illus. http://www.springer.com/economics/game+theory/book/978-3-540-68436-7 ; Springer-Verlag New York, LLC 2008

Ariely, Dan; Predictably Irrational: The Hidden Forces That Shape Our Decisions; Harpers Collins 2008

Bensoussan, Babette E.; Fleisher, Craig S. Analysis Without Paralysis: 10 Tools to Make Better Strategic Decisions;Pearson Education 2008

Bonini, Charles P., Warren H. Hausman, Harold Bierman. Quantitative Analysis For Management. 9th ed. Boston, MA: Irwin McGraw-Hill, 1997.

Bodily, Samuel E., Robert L. Carraway, Sherwood C. Frey, Jr., Phillip E. Pfeifer. Quantitative Business Analysis: Text and Cases. Boston, MA: Irwin McGraw-Hill, 1998.

Evans, James R. Statistics, Data Analysis, & Decision Modeling. 3rd ed. Upper Saddle River, NJ: Pearson Prentice Hall, 2007

Gladwell, Malcolm, Blink - The Power of Thinking Without Thinking, Litle, Brown and Company, NY, 2005

Harrison E Frank; The Managerial Decision-Making Process; Houghton Miflin Harcout, 1998

Heuer, Richard J; The Psychology of Intelligences; 1999; Center for the Study of Intelligence; CIA; www.odci.gov/csi

Holloway, Charles A. Decision Making Under Uncertainty: Models and Choices. Englewood Cliffs, NJ: Prentice Hall, 1979.

Riabackel, Ari; Managerial Decision Making Under Risk and Uncertainty; AENG International Journal of Computer Science, 32:4, IJCS_32_4_12; Nov 2006 http://www.iaeng.org/IJCS/issues_v32/issue_4/IJCS_32_4_12.pdf

Sewell, Martin; Decision Making Under Risk: A Prescriptive Approach; mvs25@cam.ac.uk; University of Cambridge; Behavioral Finance & Economics Research Symposium – 2009; Chicago 23-25 September 2009 http://www.cs.ucl.ac.uk/staff/M.Sewell/DecisionMakingUnderRisk.pdf

Cultural Intelligence: From Buzz Word to Biz Mark

Valérie Berset-Price
Valerie@professional-passport.com

Abstract

In the software engineering field, building teams across geographical regions is old news. Indeed, most software engineers are used to having counterparts located in different hemispheres. What engineers the world over are still struggling with, however, are the cultural differences that constantly interfere with the deliverables.

The truth of the matter is that few engineering schools spend enough time training students in the cross-cultural challenges and skills expected of them on the job once they become part of an international team. Universities, as corporations, underplay the power of cultural differences, not realizing that most problems encountered by international teams while working on a project often lie in the lack of cultural intelligence that professionals display.

As such, this paper explains the seven cultural differences humans experience when engaged in cross-cultural endeavors, as well as other variables professionals will encounter in their quest of the global mindset. Having a solid grasp of those differences will allow the technical expert to detach himself from the problem and look at it with a newly acquired set of eyes. Armed with new tools, the professional will now be able to dissect the problem and examine exactly how much of the dissonance is due to inattentive cross-cultural communication and unexamined cultural assumptions that are seldom universal.

Biography

Valérie Berset-Price is the President and Founder of Professional Passport®, a consulting firm located in Portland, OR, that specializes in international trouble-shooting and cross-cultural mediation for companies invested in doing business on an international scale. Valérie is a dual citizen of the U.S.A and Switzerland. She speaks five languages; has lived all over the world; and worked for Swiss, Taiwanese, South African, Brazilian and U.S. companies before starting her own practice. Her extensive background in international business development and intercultural training makes her an expert at pinpointing why global projects fail. Her training program, "Professional Passport®: Work Anywhere with Confidence," brings the global business world into focus, bridging cultures to succeed in today's marketplace. Valérie is an international presenter and a lecturer at Portland State University, University of San Diego, and Oklahoma State University. She is an international business expert for The Huffington Post *and a frequent contributor to* Training Magazine*. She holds degrees in international business from Switzerland and in International Relations from the Monterey Institute of International Studies*

Copyright: Valérie Berset-Price, June 15, 2012

Introduction

For many years in the United States, students, professionals and executives alike have looked at the field of cross-cultural communication as skills that did not deserve much attention. To them, it was the cherry atop the cake rather than the steel mold that would shape the cake. Globalization has, however, brought the world to the desk and speakerphones of many professionals who are in hourly contact with individuals born and reared (and often still located) on the other side of the earth. The result of having thrown people from different cultural background together randomly and without proper training is too often experienced as a loss of efficiency and a communication gap that creates unpleasant friction and unhealthy frustration.

Throughout this paper we will outline the most important cultural dimensions encountered in an engineering setting and provide the tools for professionals to self-diagnose whether cultural differences might be causing the problem. This document includes an introduction to the field of cultural intelligence, starting with a glossary of culture and the components forming culture. It provides access to a questionnaire (see Appendix A) that should allow the participant to identify his or her cultural DNA. We will then dive into the seven cultural dimensions that are too often erroneously considered universal to realize that they can strongly vary from country to country. Lastly, we will examine the traits that form a solid global manager: someone capable of building relationships across cultures to ensure cooperation and success for the project. Each section ends with three questions that will allow the reader to reflect on the level of preparation and training in place in his or her work environment.

1 Glossary of Culture

1.1 Culture Defined

Central to the field of American anthropology, "culture" in the 21st century is defined by four fields: biological anthropology, linguistics, cultural anthropology, and archeology. Cultural anthropology is central to this paper and was defined in the 19th century by Sir Edward Burnett Tylor[1] as "a diverse set of activities characteristic of all human societies."

In 1973 Professor Geert Hofstede[2] published the first comprehensive study of how values in the workplace are influenced by culture. Using Hofstede's definition, we can say that culture is defined as the "collective programming of the mind that distinguishes the members of one category of people from another." As such, people who belong to a particular culture (such as the Japanese culture, the Chinese culture, the French culture, etc.) have been programmed since birth to behave within the acceptable norms of that culture.

1.2 Cultural Competence Defined

Cultural competence, according to the brilliant British linguist and cross-cultural expert Richard Lewis,[3] is "a set of aligned behaviors, attitude and policies that come together in a system, organization, or among professionals and enable each to work effectively in cross-cultural situations." As such, cultural competence classes focus on the mechanics of cultures, teaching people what must be known and respected in a culture (cf. in Muslim cultures, avoid showing the sole of your foot by crossing your leg, avoid using your left hand, etc.) to prevent offending or at least committing a serious faux-pas.

1.3 Cultural Intelligence Defined

According to the work or Dr. David Livermore,[4] cultural intelligence is "an individual's capabilities to function, interact, and manage effectively in culturally diverse settings and backgrounds." Indeed, long gone is the time where populations in the West would be homogenous and work teams are formed strictly of one single ethnic group. Today's search for optimization, efficiency, profit, international market shares, and know-how often regroups people from around the world under one virtual roof.

Communicating among people who grew up around us and share similar values and expectations is not always easy. In fact, when listening to colleagues, friends and family, it seems that inadequate communication among people is at the root of most divergences and conflicts. Adding cross-cultural differences to the mix (such as differing concepts of punctuality, autonomy, independence, liberty, fairness, entitlement, merit, and respect) tends to complicate communication to the point people disengage from the conversation or bring value judgment to the equation: judgment that is rooted only in their own cultural experiences and expectations.

In the 21st century's business world, as noted in a special issue of *The Economist* on "The Future of Work",[5] leading with cultural intelligence and thus being able to adapt one's discourse, expectations and behavior based on the culture of one's interlocutor is the differentiating factor that separates success from failure. To reuse Livermore's metaphor, cultural intelligence is realizing that each time we interact with someone from a different culture, we are driving on the other side of the road. As such, we must feel the need to focus more and be fully present within the cultural differences we experience.

In view of the above, the objective of this paper is to transform software engineers into "global bridgers:"[6] professionals who know how to draw from other cultures while being cognitively aware that their own culture may interfere with their innate ability to understand someone's actions, analyze a situation, or resolve a problem. "Global bridgers" are people who have been trained to recognize how their own cultural DNA positions themselves with regard to accepted behaviors and expectations. As such, they are able to refrain from immediately condemning behaviors that may differ from the expected ones and instead focus on seeking the sheer intention behind the action. The training natural "global bridgers" underwent often started with the learning of a foreign language and a series of long periods living abroad. That said, there is more than one way to become a "global bridger." Many monolingual and mono-cultural professionals have been able to develop the level of sensibility and awareness required to lead with cultural intelligence by investing time in understanding how cultures are formed, what components are considered cultural, and how cultural dimensions vary from culture to culture. As such, they are able to readjust their lenses on a constant basis to seize the diversity global interaction brings to their desks.

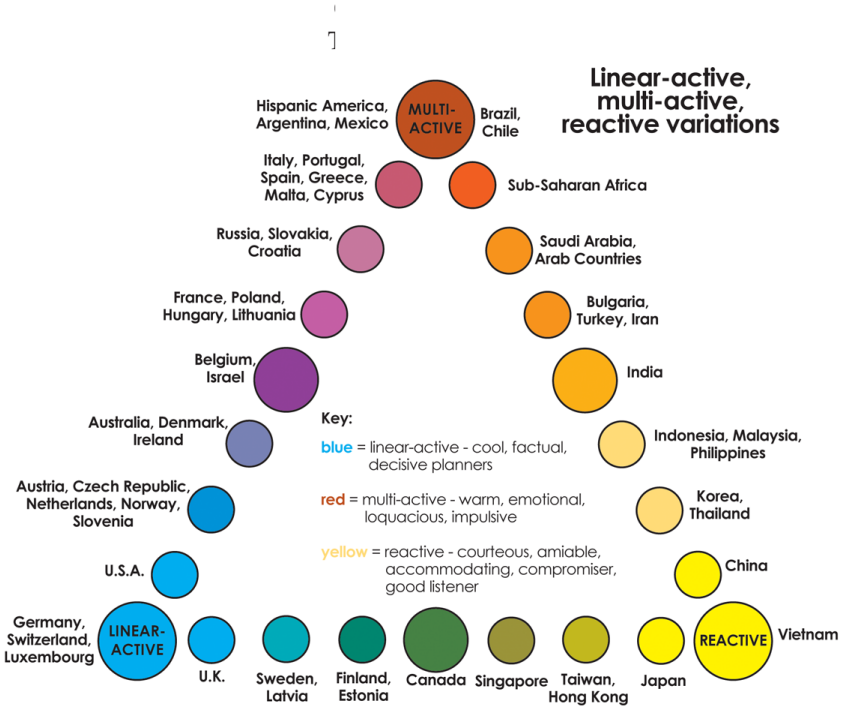
To this effect, we would like this journey to start with the completion of a questionnaire that should allow software engineers to become aware of their own cultural DNA. Please turn to Appendix A to complete the questionnaire.

The answers from the questionnaire should allow the respondent to situate him- or herself throughout the rest of this presentation as each cultural dimension is analyzed and positioned on a map of the world. Having access to our own cultural DNA helps us realize we are made of frames of references. Those frames of reference, which are rooted entirely in the culture in which we grew up, are (according to Alan and Barbara Pease[7] [2004]), fully instilled in us by the age of seven. Indeed, by that tender age most human beings know what is right from wrong, permitted and reprimanded, allowed and forbidden. By the time people join the workforce, those cultural concepts have been enforced to the point that they are ingrained and can barely be dissociated from the genetic makeup of the individual.

2 Applied Culture and Tools

2.1 The Lewis Model of Cultures

The Lewis Model[3], focuses on the different reactive variations cultures display with regard to situations and decision-making processes. It is a visual tool that should allow software engineers to quickly remind themselves that people who come from cultures different than their own might react differently to a situation than anticipated. The Lewis Model is a handy tool that teaches us at a glance that one must acknowledge cultural differences before engaging with others:



Source: http://www.crossculture.com/services/cross-culture/

The optimal way to use the Lewis Model is by reminding us of the cultural differences each country brings to the equation. Start by finding yourself on the diagram. Are you part of a large circle or a smaller circle? What color represents your country? The colored triangle, through the three large circles, illustrates three cultural types going from the linear-active (blue,) where people mainly rely on facts and logics to make decisions to multi-active (red,) where people mainly rely on emotions and impulses to make decisions; to reactive (yellow,) where people tend to accommodate the decisions making process.

As a U.S. citizen, you would be in the blue, which means you are a “linear active, cool and factual, decisive planner.” As an Israeli, you would be in a medium purple: thus still linear, but not so cool and factual as a U.S. national. If today you were scheduled to interact with a person from Turkey, we would encourage you to find where Turkey is located on the diagram with respect to your own positioning. You will see that Turkey is in orange, thus between the red and the yellow. Culturally speaking, that person will be much more emotional in his or her decision-making than you will as a U.S. citizen or an Israeli. He or she will be focused on building a good relationship with you before being comfortable with moving on to the details of the transaction. The Turk will be polite and avoid disagreeing with you, which as we will see later does not automatically mean that he or she agrees with what you will be proposing. He or she will also probably not perceive time management the same as you do. As such, and in a couple of seconds, you already have a visual of how your interaction with that person might unfold. This gives you

the opportunity to quickly assess and then adapt your own behavior and expectations to meet the ones of your interlocutor to build trust.

In addition to being concise and handy, the Lewis Model is the rare tool available in the field of cross-cultural studies to include Sub-Saharan countries, as well as certain Latin American countries that are either under U.S. embargos or considered unstable by the U.S. government.

Most research presented below in this paper comes from the debriefing of U.S. nationals working internationally. Because doing business with Sub-Saharan Africa and certain Latin American countries (such as Peru, Ecuador, Bolivia, Colombia, Venezuela) is relatively new for U.S. businesses, data on cultural behaviors are difficult to find. This explains why the subsequent maps of the world and listing of countries used to illustrate this paper tend to leave the African continent and some parts of Latin America blank due to insufficient data on the subject.

To this end, we have included a map of colonial Africa that shows to some extent the differences encountered on the continent regarding language and religion (see chapter 2.3) and how those differences tie to the former European colonies. Some careful parallels can be drawn from the historical facts, even though powerful tribal differences are entirely ignored when contemplating the continent from that angle.



Source: http://empathosnationenterprises.com/Consulate/EN-Library/Black-Studies/afcolony.html

2.2 The Role of Religion in Culture

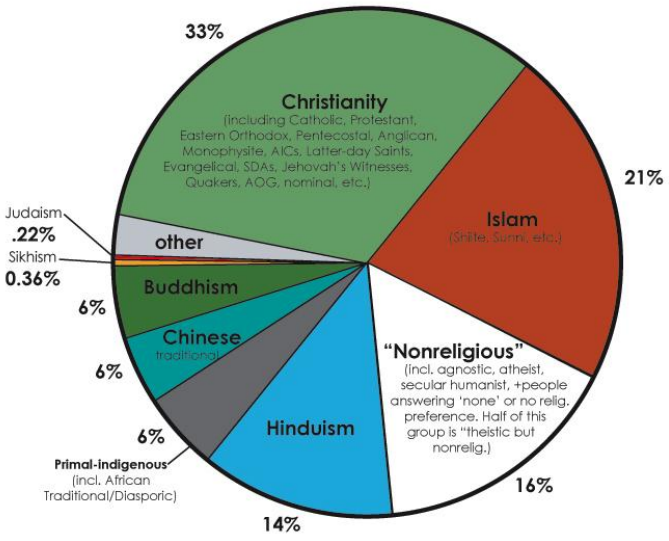
To some, religion cannot be considered evolutionary. To others, religion is the evidence that societal norms needed to be imposed for people to peacefully coexist. As such, archeologists have been able to identify that, at a time when people started to gather *en masse* in urban areas while being illiterate and uneducated, religious sanctions subtly convinced people to follow certain rules that otherwise could have put the health and longevity of many in jeopardy. Several examples of this can be found in the Muslim, Hindu, and Christian religions where people have been advised to refrain from certain activities to avoid incurring the wrath of their God.

In the Muslim world, at a time when meat was sold while exposed to the elements for days, pork was found to have the shortest shelf life, poisoning people who had the misfortune of eating the decomposing flesh. Making it *haraam*, or sinful in the eyes of Allah, was powerful enough to prevent people from eating pork.

In India, where tuberculosis (the White Plague) is still rampant, not removing one's shoes when entering someone's house is a religious offense. Infected people tend to spit in the streets, making it easy to inadvertently step in contaminated waste. Shoes are thus considered too dirty to be admitted in the house because they bring with them the possibility of contaminating the entire household with disease. Removing one's shoes before entering a house is thus a rule of conduct that has been included in the Hindu Scriptures to protect people from their own inadvertence.

In the Christian world, the Ten Commandments (or Seven Commandments for those who follow the New Testament) are also rules of conduct to maintain harmony between family members, neighbors, and even enemies. Those rules were put in place to prevent citizens from being at each other's throats when forced to live in cramped urban quarters.

Per the graph below, one can see how religiously diverse today's world is, accounting for nine distinct groups:



NOTE: Total adds up to more than 100% due to rounding and because upper bound estimates were used for each group.

Source: www.adherents.com

Now that the fabric and importance of culture in the world is clear, it is time to look at the reasons why leading with cultural intelligence in the world of global engineering are of high importance and interest.

3 Importance of Cultural Intelligence in the Global Engineering Context:

Because of the precision required in the engineering world, the interconnectedness of the users, and the multilevel approach to problem solving, it is of utmost importance that software engineers who evolve in multicultural teams develop an acute cultural awareness. Through descriptions, explanations, illustrations, and examples to the key cultural components that come into play when engaging across cultures, this cultural awareness can be developed. Through this exercise we hope software engineers will develop the skills needed to effectively manage across cultures and notice the positive impact their newly created awareness will have on projects, thus reducing failure risk considerably.

The cultural dimensions that will be examined are:

Social Stratification:	Hierarchy vs. Egalitarianism
Independence Level:	Individualism vs. Group -Focus
Business Focus:	Transactional vs. Relationship-Oriented
Communication:	Direct vs. Indirect
Concept of Time:	Linear vs. Circular
Handling Change:	Change Oriented vs. Change Resistant
Work and Life Balance:	Career Oriented vs. Joie-de-Vivre

The colored maps and country cataloging per cultural dimension are based on the research of Solomon & Schell,[8] Geert Hofstede,[2] Richard Lewis,[3] and the author's experience as an expatriate for more than 20 years.

3.1 Social Stratification: Hierarchy vs. Egalitarianism

Mark is a U.S. software engineer with a Bachelor's Degree working for a large multinational on the West Coast of the United States. His team is international and includes several Germans and Romanians. Mark has been with the company for seven years. Because he is bright, efficient, and does good work, Mark's ranking is similar to the one of Dr. Diet, his German colleague who holds a Ph.D.

According to Mark, Dr. Diet makes regular comments to him about his status, appalled that a man with such "limited" education has reached the same status level within the company that he has. While Mark finds Dr. Diet's remarks amusing, Dr. Diet is far from amused. This dichotomy creates invisible but palpable friction within the team: Dr. Diet has a sense that he is above the team and should not be expected to work so hard as the others who, in his eyes, are of lesser status. There is thus a detectable discontent that erupts from the way cultures reward achievements. This cultural difference is interpreted as a lack of fairness by the German national and, as a result, he has a potential lack of trust, confidence, and respect for Mark's professional abilities. In fact, Mark believes that Dr. Diet, the team leader, refuses to use any tool developed by him in the United States. Mark believes Dr. Diet finds his work inferior to that done in Germany by highly qualified professionals such as himself. According to Katrina, Mark's boss, the team is dysfunctional at best and does not work well together, which is reflected in the quality of the work the members do jointly.

This situation is common and is rooted in the way the two cultures operate: Germany is hierarchy and prestige-oriented (illustrated on the map below in gold), and the United States (on the map shown in red) is oriented toward equality. As such, in the United States people are rewarded based on merit and personal accomplishments rather than their level of education and social status. In Germany merit has little to do with success, as the culture is mainly oriented toward a strict hierarchy that yields rewards for

people who have carefully climbed the precise steps of the professional ladder outlined for them since birth. Those steps are coated with a strong education and many years of seniority. As a result, the Germans harbor a disrespect for the self-made man who skipped the painful academic steps, choosing instead the risk of demonstrating raw capacities on the job.

Looking at the map, we notice that these are the countries where egalitarianism is applied:

- Australia, Canada, Denmark, Israel, the Netherlands, and New Zealand being the strongest
- Finland, Norway, Sweden, and the United States being just one notch below

Hierarchy and egalitarianism is a balancing act in:

- Argentina, Belgium, France, Ireland, Poland, Qatar, Russia, Switzerland, Ukraine, and the U.K.

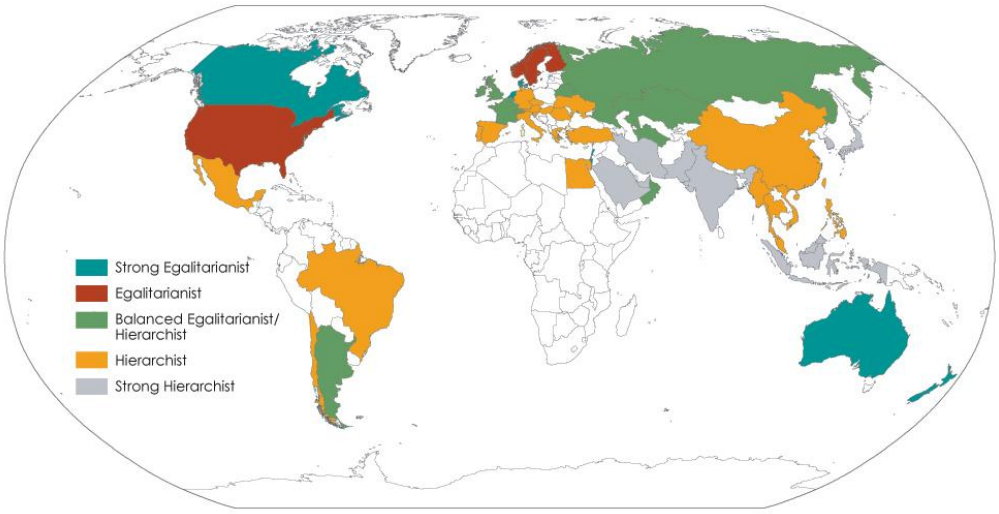
In these countries, the culture is more hierarchy-based:

- Austria, Brazil, Chile, China, Czech Republic, Egypt, Germany, Greece, Hong Kong, Italy, Mexico, the Philippines, Portugal, Romania, Singapore, Spain, Taiwan, Thailand, Turkey, Vietnam

Hierarchy is in full force in:

- Brunei, India, Indonesia, Iran, Japan, Malaysia, Pakistan, Saudi Arabia, South Korea, United Arab Emirates

GLOBAL VIEW OF SOCIAL STRATIFICATION:



Source: based on the work of Hofstede, Lewis, Solomon & Schell, and Berset-Price

In India, a country where hierarchy is tangible, the U.S. technique of promoting someone through the ranks while hosting an office on Indian land is perceived not only as despicable but somehow dangerous for the person who has been selected for promotion. In a country where the caste system defined a family's condition for generations, promoting someone without any concern for the person's gender, marital status, social status, education, seniority, ties to the community, and other members of the group

can create regrettable tensions within a company. A person who has been singled out for a promotion to the detriment of his peers may have no option but to leave the company.

It is thus of utmost importance to spend time understanding and respecting the cultural differences at play. One easy way to realize the type of culture into which the professional is stepping is through keen observation. Before making any decision, professionals should take the time to observe their surroundings and ask questions of many different people. They could find someone in-country who has a similar level of responsibility and build a strong relationship so questions can be asked in a safe way. No one should make assumptions or jump to conclusions. What the naked eye can see is often only a fraction of the reality.

Diagnosis:

- In your company, have employees and managers been made aware of the cultural differences that exist between them and their foreign colleagues as far as hierarchy vs. individualism?
- Has your company taken the time to build teams by paying close attention to the fact that the way professional reward is structured might differ in your country compared to that of the home country for some foreign team members?
- Does your company provide cross-cultural communication, management training, and coaching for managers and employees who are part of international teams?

3.2 Independence Level: Individualism vs. Group Focus

Gregg is a wonderful manager. He is a great communicator and really cares about the people who report to him; everybody agrees on that. He is supportive, caring, and always ready to roll up his sleeves when the job requires his help. Last month he was especially pleased with one of his South Korean employees. The client in Seoul was threatening to cancel the contract due to some recurring software errors that kept causing major problems within the company. Dae-Ho, known to his Western counterparts as Sam, saved the day. He not only resolved the problem, but he demonstrated great initiative; the client's complaint came during the night in the U.S., and he took it upon himself to visit the client and provide the needed reassurance that the problem had been permanently fixed. When Gregg came to work, he found an email from Dae-Ho outlining what had been done that day. To show his appreciation, Gregg sent an "atta-boy" email to Dae-Ho, copying the entire company on it. In his email, Gregg outlined how much he appreciated the initiative Dae-Ho took and how others should be inspired by it. The company needed more of that, especially in Asia. The next day, Gregg was informed by the Seoul office that Dae-Ho had presented his resignation letter to the executive team. The "atta-boy" email embarrassed him to the point that he could not continue to work there. Gregg was puzzled and furious. He requested Dae-Ho's cell phone number to talk to him in person, but Dae-Ho refused to take his calls. It was a dead end. How can a compliment so well deserved turn into a crisis where a talented employee feels forced to resign?

The truth is that Gregg is a U.S. national who grew up in a culture that fosters and rewards individualism. In the U.S. children are taught that they must be good team players, something that is largely implemented through team sports, while knowing that the success ultimately rests on an individual's shoulders. Most U.S. nationals grow up in a competitive setting where there is always a winner and a loser. In contrast, in South Korea (as in most Asian countries) the philosophy of Confucianism, a rigid ethical and moral system, teaches that the whole is only as good as the sum of its parts. As such, nobody deserves special praise because achievement depends on the efforts of many invisible hands that contribute to the success of the group–never on one single individual. Singling out someone in a group-oriented culture will make that individual appear extremely selfish, disconnected, and disrespectful of the rest of the group.

The author, on her first trip to Taipei in 1997, experienced this situation first hand: she hired Liu, a Mandarin-English interpreter, to assist her with her business meetings. Liu was a delightful young lady

with impeccable English. For a week Liu went far beyond the call of duty, something that did not go unnoticed. At the end of the week, Valérie went to the office with Liu to pay for her services and had the opportunity to meet the owner of the language bank. For about 10 minutes Valérie praised her interpreter, asking Liu to interpret what she was saying in Mandarin to her boss. The more Valérie praised Liu, the more Liu showed embarrassment. Thinking that verbal praises were not enough and that Liu expected a good gratuity, Valérie reached into her purse and gave Liu $100 as a tip. Liu immediately returned the money and excused herself. Valérie left in a taxi, puzzled by Liu's behavior; she had not even said goodbye to her. Back at the office, Valérie sent several notes to Liu, but none were answered.

On a flight to Japan two years later, Valérie sat next to a Taiwanese lady who then lived in California. They became engrossed in a conversation about cultural differences between their respective home countries (Valérie is Swiss) and the U.S. Valérie told her about her experience with Liu, aware that she might have done something wrong. Her seat companion confirmed that forcing Liu to speak of herself to her boss in a complimentary fashion would have been horrid, and giving ***her*** a big tip definitely tipped the scale as far as mortifying behavior for Liu.

Cultural faux-pas are often realized too late, once others have been offended. It is thus recommended you enter any new culture with a humble attitude, asking for forgiveness and correction if any inadvertent cultural mistakes are committed while in the host culture. A humble attitude and the desire to learn go a long way!

Looking at the map, we notice that the following countries are hierarchy-based:

- Brunei, Indonesia, Japan, Malaysia, Qatar, South Korea, Vietnam
- Austria, Brazil, China, Egypt, Greece, Italy, Mexico, Pakistan, the Philippines, Saudi Arabia, Singapore, Taiwan, Thailand, Turkey, United Arab Emirates

In the following countries, hierarchy and group focus is a balancing act:

- Argentina, Chile, France, Germany, Hong Kong, India, Iran, Ireland, Israel, Portugal, Romania, Russia, Spain, Sweden, Ukraine, the United Kingdom

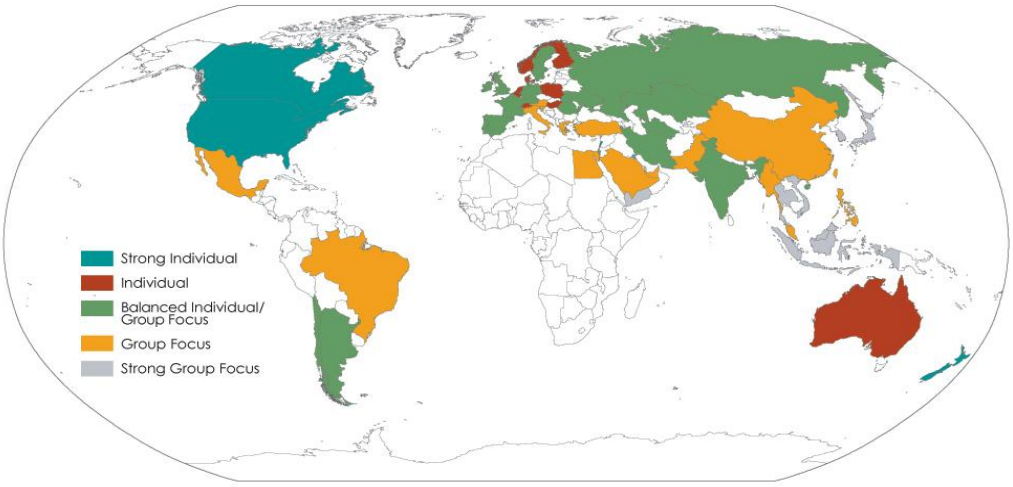
In these countries, the culture is more individualistic:

- Australia, Belgium, Czech Republic, Denmark, Finland, the Netherlands, Norway, Poland, Switzerland

Individualism is in full force in:

- Canada, New Zealand, the United States

GLOBAL VIEW OF INDEPENDENCE LEVEL



Source: based on the work of Hofstede, Lewis, Solomon & Schell, and Berset-Price

To people who come from individualistic cultures, the group-focus approach is often perceived as unfair; they tend to believe that everybody would want to be acknowledged as the person who saved the day or deserves accolades. In fact, this perception is culturally biased and people who were reared in a group-focus culture find it petty and disconnected to demand recognition, as they see the big picture and realize there is normally more than one person behind any success. It is thus of utmost importance to pay careful attention to that cultural paradigm and not assume that everybody would want personified reward. Furthermore, if a manager insists on giving praise to an employee who comes from a group-focus culture, consider doing it privately instead of in front of others.

Diagnosis:

- In your company, how do you deal with giving praise and criticism to people who may not come from an individualistic culture?
- Has your company taken the time to train its managers on showing sensibility toward the group-focus preference?
- What action would you pursue if an employee were to quit because of a misinterpreted “atta-boy”?

3.3 Business Focus: Transactionalism vs. Relationship-Oriented

Marsha is a great project manager. She is focused, efficient, and task-driven. When she is at work, she is mindful of company time and does not engage in chit-chat. Her email correspondence reflects that business-oriented style as well; for instance, she reuses the email she received and inserts her response in red in the text. She never says “hello” or “thank you” and totally ignores any inquiries from others about her private life. For example, last week, Maeve from Brazil wrote the following:

Dearest Marsha,

I hope this note finds you well. It must be summer for you now in Seattle, and I saw that graduation time is in full swing all over the nation. I love seeing the graduation ceremony of U.S. universities; it's so beautiful with the flying caps… I wish that we would do something similar in Brazil. For us it looks more like a giant party with little organization. My son graduated last year as a mechanical engineer from the University of Curitiba. I only have one child. What about you, Marsha? Do you have children?

When you have a minute, would you please send me the report that was completed last week by your team? We need to update our schedule, and being aware of your deadlines would be helpful to my boss.

Thank you very much, and have a wonderful weekend.

Your Friend,
Maeve

Now, here is the response Maeve received a few hours later from Marsha:

Schedule attached

Based on this cold response to her long and kind email, Maeve's disappointment could not be hidden. She went to her boss and asked why the U.S. team is so cold and condescending to the Brazilian team, wondering if they were racists or think they are better than Brazilians. In Maeve's heart, Marsha had been classified as a cold, selfish American with whom she would never engage again. In fact, a month later Maeve received another cold email from Marsha, this time asking her for information—which Maeve proceeded to delete without any response. Communication had come to an end. Marsha re-sent her email with the subject line "Resent. A/R" but not a word added. Maeve did not respond to that one either, and her boss (another Brazilian) thought the silent treatment was justified. If Marsha could not be polite and civil, she would not get any cooperation from the Brazilian team. When the project turned out to be late and a large amount of money was at risk, Marsha complained that the Brazilian team was unresponsive and uncooperative. The members never answered emails and would not provide the information needed for the project to move smoothly. When the Brazilian team was questioned, it responded that the U.S. team was disrespectful and rude. Marsha was shocked to learn from her boss that her communication style was at the root of the project failure! When questioned, she confessed never acknowledging emails that diverted from purely work-related topics and noted that to her those comments Maeve made about her kid were "white noise" that did not belong in a company email.

The differing communication styles and the crisis that ensued are typical and extremely common between people who come from cultures as different in that regard as North and South America are. Indeed, in the U.S. business is centered around the product/service, price, warranty, marketing, etc. Seldom is it centered on forging a relationship. In fact, in the U.S. employees are often strongly warned never to mix business and pleasure.

In cultures such as Brazil's where people are relationship-oriented, business is considered a by-product of what happens when people have great relationships with each other. As such, relationship-oriented cultures need to engage on a personal level to build that relationship. Engaging in conversation such as the one Maeve included in her original email to Marsha is a perfect example of what is being taught in universities in "Business 101" classes all over Latin America. Indeed, Latin people would never consider skipping the "hello" or the typical "personal" paragraph in their communication. There is no line between business and pleasure because business has to be pleasurable in order to take place.

Looking at the map, it can be seen that only North America is so transaction-oriented. The rest of the world in large part has a need for relationship building, and the Southern hemisphere (with the exception of Australia) demands relationship-building to establish business. The fact that Australia does not follow the rule points toward another important element outlining this difference: the role of the law in Western

cultures. In North America particularly, business owners do not feel that a relationship is needed to resolve arising problems. For that, they have hefty attorneys who specialize in taking care of noncomplying customers. In fast-growing regions, previously referred to as the emerging world, the law is often corrupted and thus not equipped to tackle business matters. As a result, business owners in those regions know that aligning only with trusted people who share the same values and people who operate on the same level will save the day once problems arise. Problems always occur in business—even when it is begun with the best possible intentions. This is also the reason why initiating business internationally requires patience and determination, as it may take several years and many business trips to earn the trust of a potential customer. In contrast, in the U.S. people are quick to place orders and few request face time.

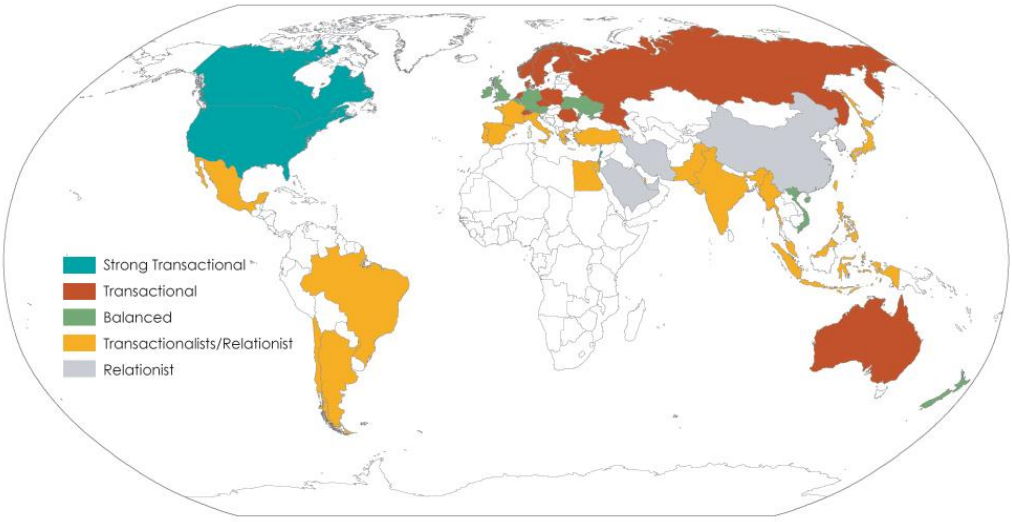
In Valérie's experience as an international trouble-shooter, being aware of that difference is the subtle element that often prevents U.S. businesses from succeeding in the international arena. North Americans do not realize that building relationships is at the core of doing business, and they often give up when purchase orders are not yielded on the second business trip.

Chinese *guanxi*, a complex social system that relies entirely on established relationships, is now fairly well known outside the country. Foreigners wanting to do business without established Chinese partnerships in China seldom see results. To many Chinese nationals, *guanxi* is the equivalent of money in the bank, as they fully grasp the importance of having solid, established relationships. As such, to the untrained North American eye Chinese employees may appear to be wasting time building their webs, as in the U.S. relationships will take you only so far due to policies related to nepotism and the belief that business and pleasure should never mix.

Lee is Chinese-American; he grew up in San Jose, Calif., does not speak Mandarin, and is third-generation Chinese. As a result, Lee looks Chinese but acts like an American, something that he and his employer neglected understanding when they sent Lee and his family on assignment in Guangdong, China. There, Lee took over a brand new Chinese team and directed it like an American would, without any understanding of *guanxi*, hierarchy, group focus, etc. The fact that Lee looked Chinese confused his team even more and made Lee's cultural faux-pas entirely inadmissible. How could he not know that in a meeting you don't single out people who don't perform? Lee also despised the bureaucratic and hierarchic Chinese system and refused to address people using their titles and names. He gave Western names to his team instead of learning how to pronounce people's name in the dialect of the region. It did not take long for Lee to realize that his team did not respect him. What he could not understand were the reasons why. He had been an acclaimed manager in California for many years and enjoyed great success. What could be so different here? The universal language of logic did not differ, and his style had not changed.

The fact that Lee's style did not adapt to his environment was compounded by his Chinese look and made the two factors a deadly professional combination. Lee came back to the U.S. eight months into his three-year assignment. Moving Lee's household from the U.S. to China twice in one year, in addition to having to send someone else to Guangdong to replace him, tallied near $100,000 in logistics cost! For a regrettable reason, Lee's managers bypassed his cultural training, leading to a costly fiasco and the loss of a profitable project.

GLOBAL VIEW OF BUSINESS FOCUS



Source: based on the work of Hofstede, Lewis, Solomon & Schell, and Berset-Price

Looking at the map, it can be seen that the following countries are transaction-based:

- Canada, the United States
- Australia, Czech Republic, Denmark, Finland, the Netherlands, Norway, Poland, Romania, Russia, Sweden, Switzerland

In the following countries, transaction and relationship is a balancing act:

- Austria, Belgium, Germany, Hong Kong, Ireland, Israel, New Zealand, Ukraine, the U.K., Vietnam

In these countries, the culture is more relationship-oriented:

- Argentina, Brazil, Brunei, Chile, Egypt, France, Greece, India, Indonesia, Italy, Japan, Malaysia, Mexico, Pakistan, the Philippines, Portugal, Qatar, Singapore, Spain, Taiwan, Thailand, Turkey

Relationship-building is in full force in:

- China, Iran, Saudi Arabia, South Korea, United Arab Emirates

As such, it is important that international players take the time to train the team to mirror the behavior of the international counterparts, even if at times it may feel counterproductive or inefficient. Being able to meet their human needs will provide deep cooperation down the line, as a well-established relationship often means devotion and loyalty to the relationship-oriented individual.

Diagnosis:

- In your company, must everyone comply with a one-size-fits-all mentality where the U.S. mindset supersedes all others?
- Is your executive team aware of the differing business focus and thus able to pinpoint the problems that arise when it is not properly taken into consideration?
- What does management do at your company to promote the global mindset among people?

3.4 Communication: Direct vs. Indirect

Egbert is the kind of man who doesn't mince words. He says what he has to say regardless of who is in the room, and he never avoids a conflict. He, in fact, *loves* conflicts and heated debates. Egbert is Dutch. His U.S. colleagues have learned that Egbert's opinion will be shared with the group, even (and especially) when unsolicited. Many on the team find Egbert abrasive and overbearing. To everybody's horror, Egbert called the V.P. of Engineering on the carpet a few months ago, and just about everyone was hoping Egbert would be fired. Interestingly, the V.P. did not seem to mind—and it can be traced to the fact that the V.P. spent several years working in Europe, where he got used to that direct communication style. In fact, the V.P. knows that in Europe being direct and not sugar-coating communication is perceived as a sign of integrity and a compliment to management. It shows that the employee is engaged and willing to contribute.

Unfortunately, the V.P. is the only one who has had the opportunity to develop a global mindset. Everybody else on the team is stuck on how rude Egbert is, and people begin to sabotage his work or avoid projects that include him. This is especially true for the Indian and Japanese members of the team. Egbert's directness shocks them. They see nothing redeeming in being so frank on purpose. It is true that Egbert does not display much of a global mindset either. He pushes his Asian team members' buttons on a daily basis by creating unnecessary confrontations, expecting a "no" instead of a "maybe" and a "yes" instead of a "might be." The Northern Europeans and the Asians on the team are like oil and water, both hoping the other will quit being who he or she is.

The truth is that everyone communicates differently based on the culture of birth; in low-context cultures, such as Europe, the orator is responsible for making him or herself understood. There is also a trend that requires that a person know good rhetoric when speaking. The message must make sense, but the delivery is just as important. Germans and French people are particularly prone to this style wherein prose is often used to make a speech or toast in front of a group. U.S. nationals, who are more to the point, tend to view the prose style as being in love with the sound of one's own voice.

Interestingly, in high-context cultures such as Southeast Asia, Japan, India, etc., the interlocutor is the one in charge of understanding the message, which is often provided through riddles, stories, and metaphors. Discovering the hidden message is a sign of cleverness and intelligence.

As a result, Mike, a U.S. software engineer who manages a large multicultural team, detests having conversations with his Asian reports. To him, most conversations tend to be noncommittal and devoid of opinion; Asians agree with everything in the meetings but will do the opposite of what they just agreed to as soon as they return to their desks. In Mike's mind, all Asians are passive-aggressive. But are they really?

Looking at the map, we notice that the following countries embrace direct communication:

- Denmark, Germany, Israel, the Netherlands, Norway, Sweden, Switzerland
- Australia, Austria, Belgium, Canada, New Zealand, the United States

In the following countries, communication between direct and indirect depends on the subject:

- Argentina, Brazil, Chile, Czech Republic, Finland, France, Ireland, Poland, Portugal, Romania, Russia, Spain, Ukraine, the U.K., Vietnam

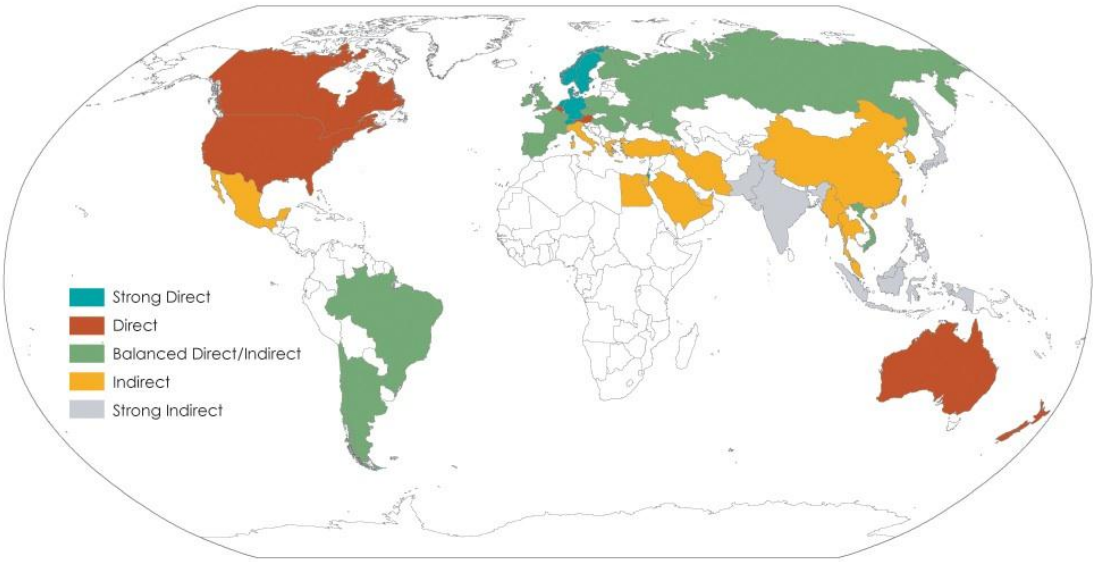
In these countries, communication tends to be indirect:

- China, Egypt, Greece, Hong Kong, Iran, Italy, Mexico, Qatar, Saudi Arabia, Singapore, South Korea, Taiwan, Thailand, Turkey, United Arab Emirates

Indirect communication is in full force in:

- Brunei, India, Indonesia, Japan, Malaysia, Pakistan, the Philippines

GLOBAL VIEW OF COMMUNICATION



Source: based on the work of Hofstede, Lewis, Solomon & Schell, and Berset-Price

In light of what we learned with regard to group-focused cultures, wanting individual commitment from group-focused individuals will often bring disappointment. Being very direct and even confrontational tends to make people shut down if they come from indirect communication-based cultures. Thus it might be better to change around that technique and instead build group consensus within the team, providing the group-focus individuals with time to regroup among themselves. When a message has to be passed down, consider involving an intermediary to deliver the message in a subtle way. This will prevent the person from losing face in front of the manager and will probably yield better results.

Diagnosis:

- In your company, do you force everybody to respond to the direct communication style favored in the United States?
- When people do not respond to your company's communication style, what technique do you use?

- Does your company move employees around, dipping them in foreign cultures in an attempt to develop cultural intelligence?

3.5 The Concept of Time: Linear vs. Circular

Eduardo is a Mexican code developer. He is brilliant at what he does, but according to the U.S. team he has no time-management skills. This makes him unreliable and too dangerous to use on time-sensitive projects. His teammates hate dealing with him because he may well work relentlessly on a project without any sleep for three days and then vanish for two. When his boss reminds him of an important deadline, Eduardo's response is either a smile or a laugh. To re-use one of his colleague's quotes, Eduardo has the "time awareness of someone who lives in prehistoric times." To that colleague, it's a miracle Eduardo graduated from college because he appears very undisciplined to his Northern counterparts who come from regions where time is linear and considered a commodity.

In several Latin American cultures (and others in gold and grey in the map) such as Mexico, Brazil and Chile, time is circular. This means that time always comes back and that people have the capacity to create time when needed. While linear cultures compete with the clock, trying to accomplish all that needs to be done within a certain period, circular cultures slice time by activity. Going to a meeting is an activity for which time has been created. Meeting a friend for a drink is an activity for which time has been created. Taking a nap or going for a walk is outside the created time and therefore has no impact on someone's schedule. It will not delay you or push you back in your schedule; it is an invisible activity. On the other hand, if a Latin or Southeast Asian person receives an unexpected visitor in his office, he or she will not inform the visitor that another activity had been scheduled. He or she will create time to be with that person while the next appointment waits.

In Latin America, where the author lived for many years, no one is in a rush and being two hours late to a meeting is considered okay. No one gets bent out of shape because the meeting did not start on time. Coffee will be served, cigarettes will be offered, other people will come in to chat: time is always put to good social use to maintain that relationship-oriented dimension that was mentioned previously. When Valérie (who comes from one of the most linear cultures in the world) worked in Chile, she had to entirely rethink her paradigms, as punctuality was just not a tool that had any weight in that culture. She often felt she had taken with her a toolbox that would allow her to fix the plumbing of a house, only to realize upon arrival that it was the computer that needed repairs. Her toolbox was inadequate and needed to be refurbished. It took her a long time to let go of her cultural assumptions and accompanying judgment calls with regard to people who could not be punctual. She had the hardest time taking unpunctual people at face value, having immediately assumed that the person was disrespectful of her and unprofessional. She realized working with many unpunctual people that her assessing tool, punctuality, was a dull blade. Unpunctual people turned out to be extremely capable and very bright; they just did not evolve in a milieu where time mattered much.

To survive and deliver her projects on time to clients in the Northern hemisphere, Valérie became good at modifying deadlines, building a three- to four-week buffer into the deliverables. As such, she could please both sides of the transaction without having to harass the unpunctual and without disappointing the on-time delivery customers. While working for a Brazilian company for eight years, Valérie noticed that punctuality and adherence to deadline were never part of her performance review. Punctuality is a dimension in Brazil that holds such little value that it is not worthy of comment!

Looking at the map, we notice that the following countries embrace punctuality:

- Denmark, Germany, Switzerland, the United States
- Australia, Austria, Belgium, Canada, Finland, Hong Kong, Israel, Japan, the Netherlands, South Korea, Sweden, the U.K.

In the following countries, punctuality is a balancing act:

- Argentina, China, Czech Republic, France, Ireland, Italy, New Zealand, Norway, Poland, Romania, Russia, Singapore, Taiwan, Turkey, Ukraine

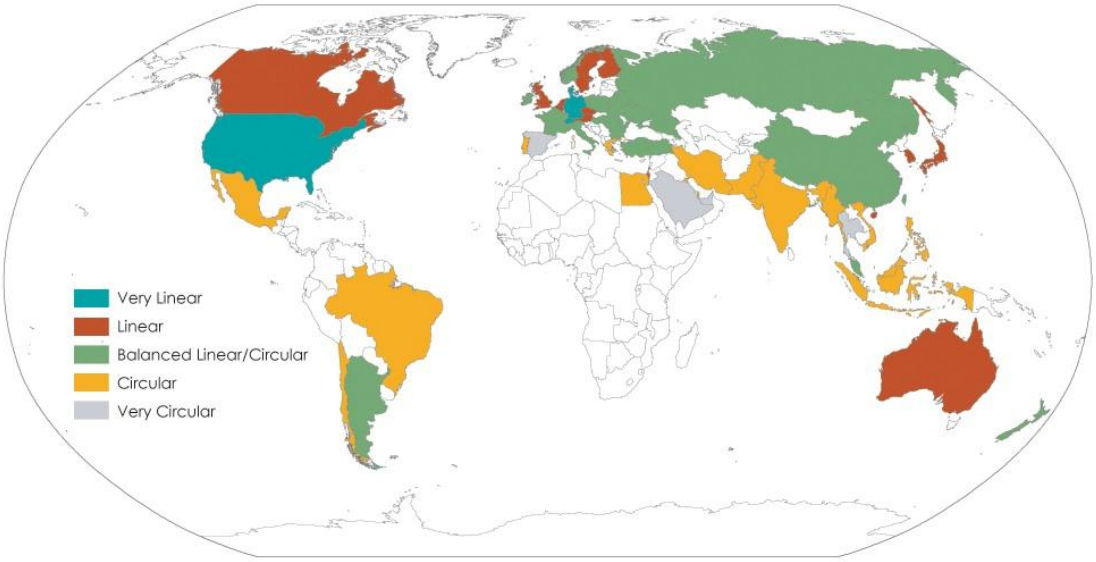
In those countries, punctuality has little meaning:

- Brazil, Brunei, Chile, Egypt, Greece, India, Indonesia, Iran, Malaysia, Mexico, Pakistan, the Philippines, Portugal, Qatar, Vietnam

Punctuality has no meaning in:

- Saudi Arabia, Spain, Thailand, United Arab Emirates

GLOBAL VIEW OF THE CONCEPT OF TIME



Source: based on the work of Hofstede, Lewis, Solomon & Schell, and Berset-Price

Without a doubt, it is impossible to change a culture. What **is** doable, however, is to change our own expectations and to manage amid cultures. As such, one must first realize and accept that there is more than one way to be a professional. My definition includes punctuality, while Eduardo's does not. My responsibility, as a "global bridger," is to manage the difference: to find ways to work in harmony as a global community. After all, performance is a bit like beauty in that it lies in the eye of the beholder!

Diagnosis:

- What does being on time mean to your corporate culture? Is being unpunctual a strike against promotion?
- In your company, do you have "global bridgers?" These are people who have the skills to work between cultures, giving you access to talents and skills that otherwise would not make the cut.
- When people do not respond to your company's expectation with regard to time management, what technique do you use?

3.6 Handling Change: Receptive vs. Resistant to Change

In the United States – a young, dynamic country – change is the essence of life. People love change and are quick to adapt to the newest technology, take in stride the most recent corporate buy-out, or even easily accept the fact that their favorite wine has been discontinued. U.S. nationals love change.

Throughout the world, the countries that have been classified as most receptive to change (Australia and Canada) are also very large and young countries that experience harsh climate changes. As such, they experience situations that are beyond humanity's control, such as endless months of snow or years of drought. Resilience and flexibility have developed out of the natural challenges one deals with in those countries, forcing people to roll with the punches. In areas where the climate is more clement, the space is more manageable, and history runs deep, flexibility and adaptability are rare. People are set in their ways and tend to resist change.

Christian experienced this first hand. As a consultant, he was hired to reconcile the IT system of several companies that were bought out by a U.S. competitor. While technically competent and fluent in French, Spanish, and English, Christian had never lived in Europe. The resistance that awaited him during his three-month stay was beyond his wildest imagination: Europeans did not want to comply with the universal IT system he was there to implement. The classes he was there to teach once the system became operational were deserted. Employees refused to show up to work in protest of the change they now had to implement in their professional lives. Yes, the IT system was operational; but nobody wanted to operate it! What a conundrum!

Christian's experience was beyond belief, and he was very poorly equipped to deal with the resistance he encountered because it came from every level of the organizations and was expressed slightly differently in every country in which he operated. In brief, while Christian spoke the languages needed to communicate with the employees, he was not cross-culturally prepared to handle the situation. He was unfamiliar with the cultures and at a loss. He and his employer had wrongly assumed that employees would embrace the change he embodied.

Looking at the map, we notice that the following countries embrace change:

- Australia, Canada
- Hong Kong, Israel, Japan, the Netherlands, South Korea, Sweden, the U.K., the United States

In the following countries, embracing change is a balancing act:

- Argentina, Austria, Belgium, Brazil, China, Denmark, Finland, France, Germany, Greece, India, Ireland, Italy, Malaysia, Pakistan, Poland, Portugal, Qatar, Romania, South Korea, Spain, Sweden, Switzerland, Taiwan, Ukraine

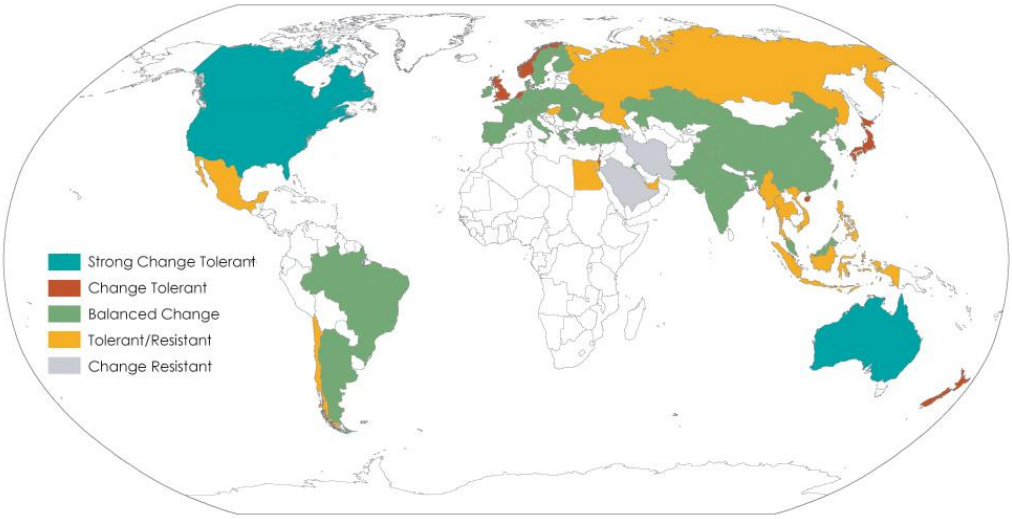
In those countries, change is resisted:

- Brunei, Chile, Czech Republic, Egypt, Indonesia, Mexico, the Philippines, Russia, Thailand, United Arab Emirates, Vietnam

Change is strongly resisted in:

- Saudi Arabia, Iran

GLOBAL VIEW OF HANDLING CHANGE



Source: based on the work of Hofstede, Lewis, Solomon & Schell, and Berset-Price

The ability to deal with change is also interestingly linked to the permission to fail. (Note: this is not the focus of the above map.) In cultures where failing is perceived as a canvas for growth, people feel allowed to take risks and engage in activities where they may not yet be in full control. In cultures where failing is a disgrace from which it is difficult to recover (the concept of "losing face"), embracing change with an open mind is harder. As such, it is essential to keep that cultural paradigm in mind, especially when hiring. Charlie, a talented high-tech Chinese national who interviewed for a manager position in Shanghai for a U.S. subsidiary, shared with me that in his interview a panel of four interviewers asked him to speak of the failures he encountered in his life and how failing helped him grow as a professional and an individual. This question made Charlie so uncomfortable that he became mute. His muteness was translated as a lack of maturity and experience for the position. He was the best-qualified candidate, but he did not get the job due to the nature of his response to the question. The panelists had no idea that asking a Chinese person about failure was the equivalent of asking a U.S. citizen about his sexual orientation in an interview setting.

Failure in many Asian cultures, including Japan, is worth committing *hara-kiri* over. There are certain questions that just cannot be asked in certain cultures, and the interviewers must be aware of them so that talents, as well as opportunities, are not bypassed by imposing our cultural views of what is right on others who have been reared with a different cultural bias.

Diagnosis:

- How do you handle change at your company?
- When people do not respond to your company's expectations with regard to change management, what technique do you use?
- Are your H.R. directors aware of the cultural differences at play when interviewing candidates from foreign countries?

3.7 Work and Life Balance: Career-Driven vs. *Joie-de-Vivre*

According to Gloria, a U.S. software engineer-turned-project manager, her frustration with her European colleagues is the main reason she resigned a few years ago from the job and went back to coding. She also switched companies, making sure her team members would be exclusively North American, Japanese, South Korean, or Taiwanese. In short, Gloria had developed an allergy to countries that are not so focused on work as she is. In her view, the expectation of her being in charge of delivering results while her counterparts were either on religious holidays, four-week vacations (during which they would refuse to check their hand devices or answer any urgent emails), or simply not interested in working on the weekend because the weekend was "sacred" to them was unrealistic. Moreover, Gloria was puzzled by the lack of interest most Europeans had toward money. In Gloria's world, no one declines a promotion that has a bigger dollar sign attached to it. Everyone understands that work comes first, including the family. In Europe, where health, education, and nursing care are part of the benefits citizens reap by paying taxes, making more money is not always that appealing to employees. All it does is push them into a higher tax bracket. Financial security is not people's concern in countries like France, Germany, and the U.K. The concern is to have a healthy life balance where people have plenty of time to enjoy life with friends and family.

Large U.S. multinationals have, however, struggled with the concept of work and life balance for quite some time now, with U.S. employees being burnt out to the point of becoming professionally inefficient and managers witnessing families falling apart due to work overload. The concept of "work-life integration" (IBM) or "work-life effectiveness" (Intel) remains a challenge in cultures where natural boundaries do not exist between the two concepts and no labor law enforces a maximum amount of hours worked.

But the U.S. is not the only country with a work and life balance problem. In Japan, it is common to read in the newspaper that another executive or employee died of *karōshi*: death by overwork. It is an honorable death that proves that the person had the right priorities in mind throughout his or her life.

In Brazil – a country that focuses on *joie-de-vivre* (the concept that life must be enjoyed to its fullest through pleasurable activities) – the obituary reads that too many nights of drinking and sleep deprivation resulted in death from *carnival*. Dying from over-indulging in the Brazilian culture is a noble death as well, showing that the person had life enjoyment in mind and lived without any regrets.

Looking at the map, we notice that in the following countries people live to work:

- China, Japan, the United States
- Canada, Australia, Hong Kong, Singapore, South Korea, Taiwan

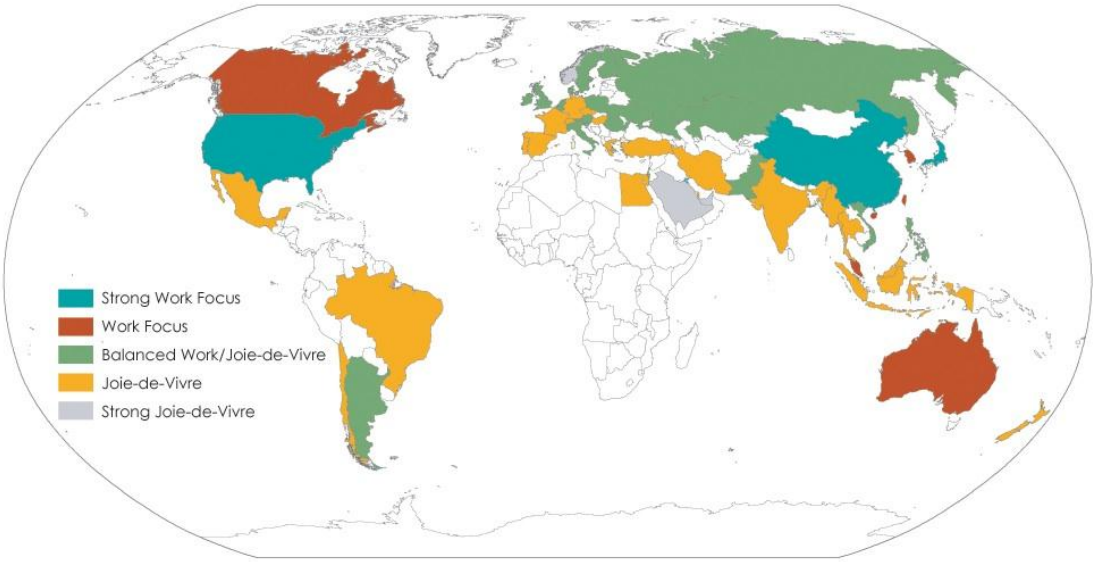
In the following countries, career and joie-de-vivre make for a balancing act:

- Argentina, Austria, Belgium, Denmark, Finland, Ireland, Israel, Italy, Netherlands, Pakistan, Philippines, Poland, Romania, Russia, Sweden, Ukraine, the U.K., Vietnam

In these countries, people work so they can enjoy life:

- Brazil, Brunei, Chile, Czech Republic, Egypt, France, Germany, Greece, India, Indonesia, Iran, Malaysia, Mexico, New Zealand, Portugal, Qatar, Spain, Switzerland, Thailand, Turkey
- Norway, Saudi Arabia, United Arab Emirates

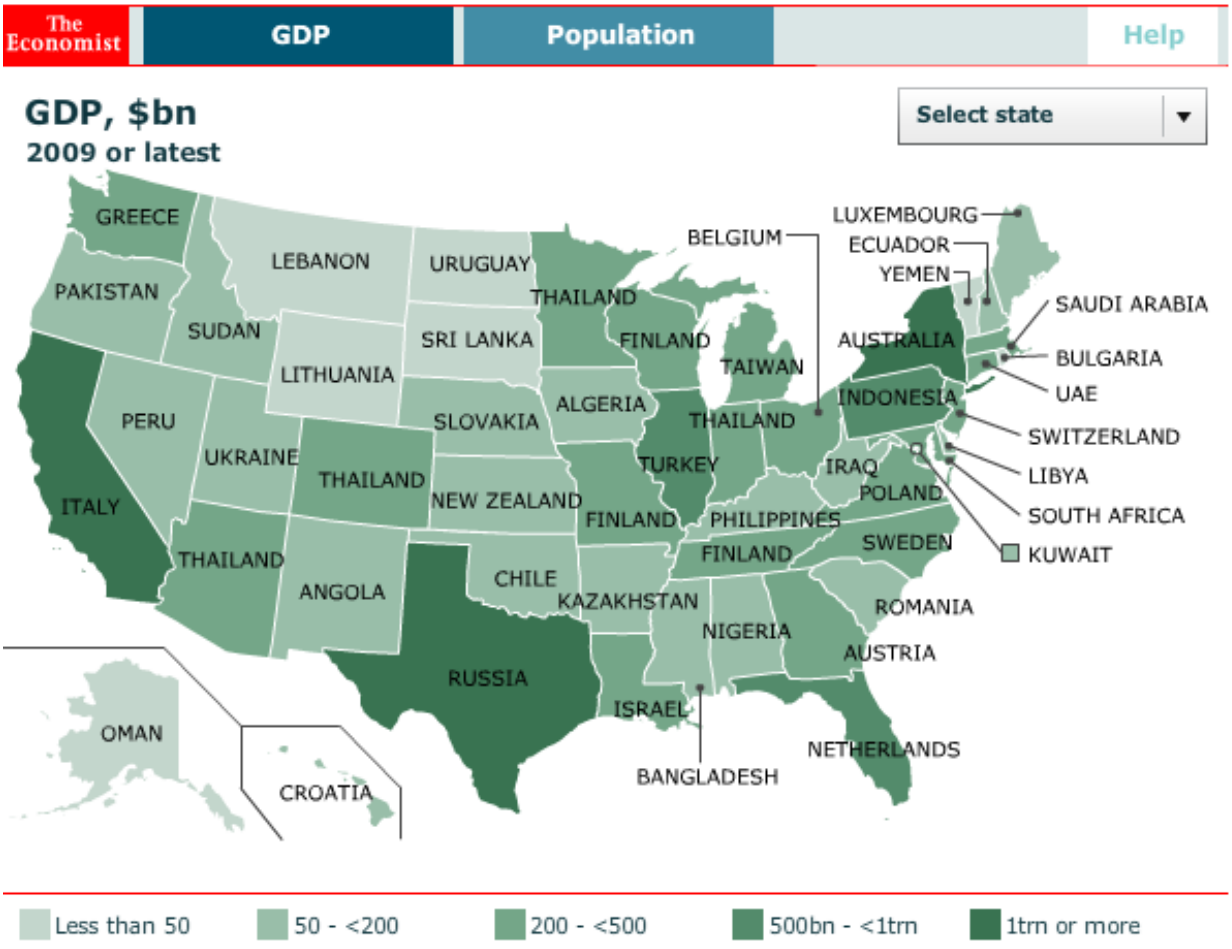
GLOBAL VIEW OF WORK AND LIFE BALANCE



Source: based on the work of Hofstede, Lewis, Solomon & Schell, and Berset-Price

In an interview with *Salon Magazine*, Thomas Geoghegan,[9] a U.S. labor lawyer in Chicago and author of a book on work-life balance, stated, "European social democracy – particularly Germany's – offers some tantalizing solutions to our overworked age. In comparison to the U.S., the Germans live in a socialist idyll. They have six weeks of federally mandated vacation, free university tuition, nursing care, and childcare." An article in the English edition of *The Epoch Times*,[10] points out "an average American full-time employee works 1,804 hours per year, while an average German works 1,436 hours. This equals to Americans working an extra nine hours every week."

That said, many U.S. nationals tend to calculate their output based on their GDP, convinced that as a whole their country is much more efficient than the rest of the world. A map compiled by and published in *The Economist,*[11] shown below, shatters that belief. Indeed, no U.S. state produces an equivalent GDP to the one of Germany.



The Economist. *Comparing U.S. States with Countries*. January 13, 2011

The challenge at hand once again comes from the limited available ability within corporations to manage those cultural differences: culturally savvy people who know how to get what is needed from employees to meet a critical deadline and to release for additional free-time in appreciation once the effort has been accomplished. Creating awareness within corporations that work and life balance will ultimately serve the company better and yield nonnegligible monetary results is important. No other successful Western country embodies that theory better than Germany.

Diagnosis:

- How is work and life balance promoted at your company?
- Are your managers aware of the cultural currents and the efficiency that derives from countries where work and life balance are well outlined?
- Will your executives be receptive to the above data, showing that efficiency can come from working less and having strict policies in place with regard to evening and weekend work?

4 The Global Manager

4.1 The Importance of Nonverbal Cues

Garrett, a software manager in charge of international teams, understood the importance of nonverbal cues when he went to India the first time. Wanting to be comfortable during his long intercontinental trip, he wore flip-flops, an old washed-out cotton T-shirt, and sweat pants on the plane. Unfortunately, upon landing in Mumbai he was informed that his luggage did not follow. The Indian manager who awaited him suggested they go shopping in the morning and buy Garrett what he would need for the coming days, which would take them outside of Mumbai. Not liking to shop and not wanting to incur any unnecessary expenses, Garrett voted against the idea of going shopping and instead asked the hotel to wash his travel attire.

To the protest of his Indian counterpart, who was wearing a suit and a tie, Garrett climbed in the car wearing his lounge clothes, thinking that anyone would have compassion for him. After all, this was beyond his control: his luggage was missing. Unfortunately, the Indian client did not show much compassion for Garrett's presentation skills. The Indian client felt offended that the U.S. vendor would not consider investing in some decent clothes to show up at a business meeting. As a result, the President and CFO did not join them for the meeting, and nothing was accomplished through Garrett's trip.

At first, Garrett could not believe that his sweatpants and flip-flops could be responsible for the cold reception he received in India. After all, in Redmond, Wash., his home base, an attire of sweatpants, shorts, and T-shirts is practically a uniform for employees and managers alike…and no one cares. What Garrett did not know is that outside the United States, the stratification of cultures is often hierarchic (see chapter 3.1). As such, managers in a country such as India are normally issued from the upper class of that culture. Many went to boarding school where they were groomed to make a perfect first impression through their nonverbal cues, aware that it does make a significant difference in the society in which they evolve.

It is thus of high importance to take the time to groom global managers and subject them to the training that will ensure good table manners, adequate packing strategies, polite communication style, etc. For example, Garrett should never have traveled in sweat pants and flip-flops. Indeed, while traveling employees represent the company for which they work and should be aware of the importance of making a good impression on their fellow travelers. Garrett could have worn a comfortable pair of trousers and a sports coat. If wearing those articles of clothing was too much for him, he should have packed them in a carry-on. With or without checked luggage, he would have been able to pull off a professional look, thereby creating value for his employer out of the expensive overseas trip he undertook on behalf of his company.

A similar experience made a competitor of Garrett's company aware that employees engaged in international endeavors needed to be groomed in cross-cultural table manners. A large contract was lost because one of the managers went to Abu Dhabi not realizing that using his left hand to serve himself off a plate would be considered the equivalent of a foreigner spitting in the soup in the United States. He concluded his stay by putting his feet on the desk, exposing the soles of his feet to the rest of the team. As a result, the contract was given to a European firm and communication with the U.S. ceased. Helping employees, especially the ones who did not benefit from living abroad and learning a foreign language, to build a global mindset and learn how to lead with cultural intelligence is essential in the 21st century.

Diagnosis:

- What selection process does your company have in place to ensure that the right people represent the company's interests across cultures?
- What kind of cultural training does your company provide to employees who enter the global arena?

- Does your company offer one-on-one coaching for employees who show frustration while engaged across cultures?

4.2 English as a Universal Language

Another great source of tension and misunderstanding comes from the assumption that anyone who speaks English at a certain level of fluency understands the cultural values that come with it. In the United States, a country formed primarily of monolingual and mono-cultural citizens with a rate of passport application of 33%,[12] not enough professionals understand the challenge it represents to non-English speakers to learn the language to a level of fluency. Professionals who come from countries where the alphabet is entirely different and where exposure to English is limited have it especially tough.

In the experience of the author, a nonnative English speaker, understanding a culture's idioms and learning its innuendos and metaphors are without a doubt the most challenging part of becoming fluent. International business people should remember that as a native English speaker, the responsibility is to limit the number of sports analogies and metaphors that are rooted strictly in the culture. For example, as of today only the U.S., Burma, and Liberia keep using the Imperial System. All others use the Metric system. As such, counting in yards is meaningless to the rest of the world.

American football is also an anomaly in the world, as other cultures play rugby or soccer rather than football. Informing a Chinese person that you expect her "not to fumble the ball," "not to just punt," "to get the first down," "to pound the rock," or "to quit dancing and run North" will probably make little sense to her, as she will not know American football rules. She will thus probably not be able to deliver the anticipated results due to your lack of clarity.

People for whom English is an acquired language will often:
- Have an accent
- Make mistakes
- Say things without realizing what they mean
- Have different cultural values
- Have different religious beliefs
- Have a hard time admitting they did not understand you

To meet non-English speakers halfway, do the following:
- Speak clearly
- Speak at a pace that is not too fast and not too slow either
- Avoid metaphors and slang
- Use common words
- Remove the contraction from your speech
- Be patient; repeat your message differently when needed
- Be grateful; realize that your interlocutor is making an effort to communicate with you in perhaps your only language
- Don't operate on assumptions
- Have a good sense of humor

Diagnosis:

- Are your employees aware of the above language-related recommendations?
- Are your employees encouraged and financially supported in taking classes in a foreign language?
- Are your employees and executives making a conscious effort to include and engage nonnative English speakers?

4.3 Grooming Global Managers

In the past 20 years, technology has advanced very quickly. Thanks to technology and especially the Internet, everyone is now able to interact instantaneously with people located on the other end of the earth with minimal (if any) difficulties. This technological ease of communication often makes everyone forget that only technology has evolved so quickly. Humans are still pretty much locked in the cultures from which they come.

The technological improvements have paved the way for a global community where tolerance and respect for each other's differences is instilled. Thomas Friedman,[13] made everyone believe a few years ago that from now on the world would be flat and that everything would be handled the same way the world over. Anyone invested in managing people across cultures will find this hard to believe. It is, however, true that travel, as well as academic and professional exchanges between countries, will become more and more prevalent and thus allow people to recognize and admit the cultural differences. As such, they will develop an awareness of others, resulting in a flexibility of the mind that will permit easier communication and greater accomplishments while remaining rooted in their own cultural values.

Admitting that becoming a global professional is the challenge of many in the 21st century, let us analyze the traits a global manager embodies, also referred to throughout this document as a "global bridger:"

Learns quickly in new context	Knows how to build partnerships
Communicates with ease	Embraces multiculturalism
Floats seamlessly between cultural dimensions	Speaks more than one language
Listens to others	Has established network in-country
Has lived abroad	Respects people's religious beliefs
Influences others	Speaks with a global voice
Knows business culture in-country	Xenocentrist

Armed with these traits and skills, global managers will earn respect from multicultural team members. In them, they will find understanding and respect for their cultures, which is at the base of any cross-cultural partnerships.

Diagnosis:

- How do you recruit global managers?
- Does your company have a policy of positive discrimination for international experience and abilities?
- Is cross-border experience part of your management technique (short-term visit, short-term transfer, long-term posting)?

Conclusion

Developing a global mindset is a journey, not a destination. Becoming a global bridger is a conscious decision that will enhance your life and the ones of others around you who come from different cultures. Actively working on developing the skills discussed throughout this paper will allow you to look at situations and problems from different angles, and ask questions around the concept of intent vs. impact that otherwise may be ignored. It is indeed impossible to master the intricacies each culture displays; they are too many to remember. It is, however, possible to work toward the goal of leading one's private and personal life with cultural intelligence by applying the simple tools discussed in this paper. We certainly hope you will find benefits to including cultural awareness in your life and that the benefit of being culturally savvy will bring richness and enjoyment to you and yours.

References

[1]Tylor, Edward Burnett. Primitive Culture and Anthropology. (Harper 1958).
[2]Hofstede, Gert. Gert Jan Hofstede, Michael Minkov, Cultures and Organizations: Software of the Mind. 3rd Edition, McGraw-Hill USA, 2010.
[3]Lewis, Richard. When Cultures Collide. (Nicolas Brealey Publishing 2006).
[4]Livermore, David. Leading With Cultural Intelligence: The New Secret to Success. (Amacom 2010).
[5]Economist, The. (SHRM 2010) *Global Firms in 2020: The next decade of change for organizations and workers*. Economist Intelligence Unit. September 2010
[6]Cabrera, Angel and Unruh, Gregory. Being Global: How to think, act, and lead in a transformed world. (Harvard Business Review 2012).
[7]Pease, Alan & Barbara. The Definitive Book of Body Language. (Bantam 2004).
[8]Solomon, Charlene and Michael Schell. Managing Across Cultures: The Seven Keys to Doing Business with a Global Mindset. (McGraw Hill 2009).
[9]Rzhevskiy, Ilya. *"Volkswagen to Improve Work-Life Balance: VW to bar emails after work hours in Germany."* The Epoch Times, English Edition. January 1st 2012.
[10]Geoghegan, Thomas. Were you Born on the Wrong Continent?: How the European Model Can Help you Get a Life. (New Press 2011).
[11]The Economist. *Comparing U.S. States with Countries*. January 13, 2011.
[12]Travelstate.gov. http://travel.state.gov/passport/ppi/stats/stats_890.html.
[13]Friedman, Thomas. The World is Flat: A Brief History of the 21st Century. (Picador 2007).

APPENDIX A

QUESTIONNAIRE: What is your cultural DNA?

1. Do I believe that the invisible social layers that form society are important and must be respected at all costs, or do I perceive that the self should have the privilege to float freely among those layers?
   - ☑ Layers must be respected. I believe in hierarchy.
   - ☑ It depends. I am not entirely against layers, but there are times I wish they weren't there.
   - ☑ Layers are unnecessary, but I am able to perform within them.
   - ☑ The self should be allowed to move freely. I am an egalitarian.
2. When at work or on vacation with a group of people, am I mostly concerned about the well-being of the group, or am I more interested in pursuing my own interests?
   - ☑ The group and its harmony are of utmost importance to me. I would never break away to do my own thing.
   - ☑ The group harmony is important, but I don't mind doing my own thing when the group doesn't support me.
   - ☑ Each person should be free to pursue what he or she wants. The group is a good place to which to return once I have achieved what I personally want.
   - ☑ I never go anywhere with a group. I do things on my own.
3. When I conduct business, do I tend to stick to business? Or do I favor building a relationship with the person with whom I am involved first?
   - ☑ Business is not about making friends; it's about making money.
   - ☑ I prefer to keep business and pleasure separated.
   - ☑ People are important to me; I like a solid and pleasant relationship before I engage in doing business with someone.
   - ☑ I don't care how much money is at stake. If I can't bond with the person, there will be no business.
4. What is my communication style? Am I a direct person, saying what I mean? Or do I tend to avoid being confrontational, favoring an indirect style?
   - ☑ I am very direct. I dislike innuendo and metaphor. I say what I mean and then move on.
   - ☑ I prefer avoiding conflict; but if I have to make myself clear, I will.
   - ☑ I avoid conflicts. If people are not smart enough to realize what they are doing, I prefer to disengage and walk away.
   - ☑ I speak through metaphor and innuendo. If I have a message to pass, I will ask someone else to do it for me.
5. Who am I when it comes to time management? Am I punctual in meeting people and mindful of deadlines? Or do I see time management as an unnecessary imposition that I often blow?
   - ☑ Time defines me. I am always on time (or early) and perceive punctuality as a sign of respect.
   - ☑ I am okay being on time when it comes to professional obligations. But in my private life, I am much looser with time.
   - ☑ My friends and colleagues have learned to deal with my inability to be on time. I am always late.
   - ☑ I don't care about time. In my culture, we create time as needed; what has not been accomplished today will be accomplished another time.
6. How do I deal with change? Is change something necessary that allows me to evolve and that I embrace? Or is change something that I resent and dread?
   - ☑ Bring it on! Change is what keeps me alive. I love to learn new things and don't mind making mistakes as I learn.
   - ⑨ Change is all right if I have plenty of time to adjust to it.
   - ⑨ I don't like change. I like to know that I am in control and that I master what is expected of me.
   - ⑨ I resent change. It brings to my life the possibility of failing and losing face. I like to keep things the way they are.
7. Do I live to work, or do I work to live?
   - ☑ I love my job. My job defines who I am. I wouldn't know who I am if I didn't have my career.
   - ☑ I accept the fact that my personal life is dictated by my professional obligations.
   - ☑ My job is important, but I must have balance between my private and professional life.
   - ☑ I go to work so I can afford all the pleasant escapes life offers. My job is a necessary evil.

Critical Factors Characterizing Projects and Lifecycle Models

Kalman C Toth

kalmanctoth@gmail.com

Herman Migliore

herm@cecs.pdx.edu

Abstract

Systems and software engineering lifecycle process methodologists boost their favorite approaches, but rarely discuss which problems these methods are best suited to solve or when they might indeed represent an ill-advised choice. From more traditional waterfall lifecycles to those popularized by the agile development community, proponents describe their benefits, often with built-in bias. Some methodologies claim that they can be customized to fit large, highly critical applications as well as very simple ones. Rarely do the claims present a balanced perspective about how the method addresses the needs of a given project, or how it can be adapted to fit the problem at hand.

The overriding aim of the work that led to this paper was to explore how critical factors characterizing projects might be applied by engineers to assist in the selection of an appropriate lifecycle process for the project. This paper summarizes the common lifecycle processes advocated by systems and software engineers, identifies eight critical selection factors for characterizing lifecycles and projects, and suggests an approach for systematically matching projects to lifecycle processes for the purpose of selecting the most suitable lifecycle for a given project. The authors have also illustrated the first step of such an approach by characterizing the most popular lifecycle models in terms of these eight selection factors.

Biography

Kal Toth has over 30 years of technical, consulting and management experience working for small, medium and large-sized technology companies including aTrust Inc., Hughes Aircraft, Datalink Systems Corp., the CGI Group Inc., the Software Productivity Centre (Vancouver, BC), and Intellitech Canada. Past Executive Director of the Oregon Master of Software Engineering (OMSE) program at PSU and for 12 years delivered systems/software engineering and project management courses at PSU, OSU, TechBC, Simon Fraser, U of Alberta, and UBC, he completed his Ph.D. in systems engineering at Carleton University (Canada), and is a professional engineer with a software engineering designation in B.C.

Herman Migliore has forty years' experience in engineering design, application of computational mechanics (part of his graduate studies and research activities), and development of design methodologies (part of his role in design education). Since 1997, he has been Director of Systems Engineering at Portland State University - an online masters program intended for experienced, practicing engineers. In his capacity as director, he has participated in many projects that apply systems engineering to a wide variety of areas for small, medium, and large industry and government sponsors.

1 Introduction

Software and systems engineers have systematically defined and progressively improved their processes, methods, and techniques over the years. However, many open issues for study remain. One particularly challenging area has to do with how to go about selecting a suitable lifecycle process (or processes) for the problem at hand [1] [2]. For example, when should one adopt a highly disciplined lifecycle, versus a much more agile process, or one that balances discipline and agility [3]?

When preparing to undertake a new project, the prudent engineering manager should ask:

- Which life-cycle processes are acceptable candidates? (Do competencies/resources fit?)
- Which process is most likely to yield positive project economies and competitive advantage?
- Will a given candidate be more or less vulnerable to project over-runs, failures, or lost business?
- Is it feasible to assess the properties of alternative lifecycles and make trade-offs among them?

More specifically, the engineering manager should look for objective evidence that a given lifecycle development model will indeed be the most suitable choice for the planned project.

The approach we have taken to tackle this problem has been a practical one – review what others' have said and done (including Boehm [3], Kruchten [8], Cockburn [9] and others) with respect to the application of popular system and software development models; observe the challenges they have encountered to enhance understanding as to what worked, what didn't, and when; and propose a strategy that could lead to practical methods and guidance for the purpose of lifecycle selection.

Our estimates are offered with some caution as they are experience-based and can only be interpreted as subjective. However, they have been derived from our collective 50-years of systems and software engineering experience including the transition of a $1B air traffic control project [10] from a waterfall lifecycle to an iterative/incremental lifecycle using Rational methods and tools, as well as numerous smaller e-commerce development projects that adopted incremental and evolutionary process models to accomplish their objectives.

Grounded on experience and subjectivity, we nevertheless expect that quantitative and empirically driven methods could emerge from additional study. Readers are encouraged to rebut the ideas we have put forward, thereby contributing to this effort to discover systematic methods for characterizing projects and selecting suitable lifecycle development models for them.

An important factor that we have not factored into our proposed approach is the enterprise's existing process culture. Certainly, the overriding factor for many smaller companies will be what they know, what they have, and what they are good at. However, it is safe to say that an increasing number of medium-sized to large enterprises, especially system-integration firms, possess the skills and know-how across many if not all of the lifecycles discussed in this paper. And nimble practitioners and learning organizations are exploring their process options going forward – progressively improving and adapting their existing processes to keep pace and beat out the competition. This paper may be of particular interest to them.

2 Candidate Lifecycle Models and Selection Factors

Our review of systems and software engineering lifecycles considered the waterfall, iterative, incremental, spiral, evolutionary and agile lifecycle development models. Collectively they cover virtually the entire range of lifecycle models in practice today if one excludes ad hoc development methods. Therefore we have constrained the scope of our discussion of lifecycle selection factors to these models.

In the context of these models we reviewed the attributes of each. After some reflection we subjectively derived our preferred (top-eight) critical factors for characterizing projects and selecting a suitable lifecycle for a given project, namely:

- Quality/Maintainability
- Application Domain
- Size / Complexity
- Requirements Uncertainty
- Progress Visibility
- User Involvement
- Requirements Volatility
- Urgency

The following paragraphs define, in practical terms, the context, meaning and implications of these selection factors.

Quality/Maintainability: Projects constructing complex systems often demand high levels of delivered quality and built-in maintainability as they are highly driven by the need to meet stringent criticality thresholds and contain total lifecycle costs. For example, customers in the aerospace and military sectors will write explicit quality and maintainability targets, standards, and documentation into their requests for proposals and acceptance criteria which must be met by the contractor. At the other end of the spectrum, solutions for experimental and concept validation purposes care little about operational performance and down-stream maintainability effects. Of course, a majority of technology projects lay on a grey-scale between these extremes.

Application Domain: The category of application is also a very significant project driver. High-end applications are those characterized by critical performance, reliability, availability, security, and safety requirements; moderate applications include business and enterprise applications; low criticality applications include proof-of-concept demonstrations, experimental prototypes, and vanilla-flavored web-sites.

Size / Complexity: For simplicity, we consider these two factors together as they are often, but not always, correlated fairly linearly. Certainly, an air traffic control system would be a highly complex and consists of a very large number of hardware components and lines of software code. Relatively speaking, a real-time robotics system would be proportionately smaller and less complex. One could assume, for example, that a planned 500K lines-of-code (LOC) software system would be considered both large and highly complex; a 10K LOC subsystem would be considered small and relatively simple; and medium-sized projects would range in-between.

Requirements Uncertainty: Many projects will commit to resources and meeting the needs of a customer before they properly comprehend the functions and features to be developed. Let us take the position that requirements are highly uncertain if they are only verbalized or if they have been expressed in a few pages with little input from users. In contrast, requirements should be relatively certain when considerable resources and schedule have been devoted to studying user needs and developing evaluation prototypes.

Progress Visibility: Visibility into progress can be achieved through demonstrations and through documentation – both have their shortcomings. Demonstrations are favored by many customers but if not supported by other measures of progress can give the illusion of more progress than is actually being achieved. Documentation is harder to translate into real progress but simplifies contracting – hence it is favored by those with accounting and legal mindsets. For this discussion we assume that ongoing and frequent demonstrations to customers and users accompanied by some useful documentation can provide effective visibility into progress. In contrast, infrequent demonstrations of functionality combined with scant documentation provide little or no visibility into achieved progress.

User Involvement: User involvement can be with respect to developing requirements specifications, validating requirement specifications, inspecting prototypes, supporting detailed design and development, reviewing specifications, and accepting released products. Users heavily committed to supporting most of these development efforts are considered to be “highly involved”. Marginal contributions from users in a few of these areas should be considered to be “low” involvement.

Requirements Volatility: This factor requires a good understanding of the problem domain, especially with respect to the maturity of the customer, users, and the development team. If the application is unprecedented or if the problem is in an emerging area, stakeholders will not be sure of their priorities and what can be feasibly accomplished. Project management and stakeholder vacillation can exacerbate requirements volatility. Often, but not always, requirements uncertainty will go hand-in-hand with requirements volatility.

Urgency / Time-to-Market: Demanding market forces will put pressure on many commercial software projects to create and release software functionalities and features as early as possible – often pushing the development to take shortcuts rather that down-scoping the project. Schedule pressure will tend to be moderated in mission-critical projects because they tend to adopt disciplined oversight and accountability measures.

3 Characterizing Lifecycle Development Models and Projects

We reasoned that it should be possible to characterize both lifecycle models and projects in terms of the above eight factors and then match a given project's characterization data to each lifecycle model's characterization data to locate and thereby select the best fit. Sensitivity analyses could be conducted by varying the project's characterization data; and trade-off studies could be conducted by estimating project costs and schedules under a given lifecycle.

We began by first exploring the challenge of characterizing lifecycle development models. Such characterization would involve assigning appropriate values ("estimates") or ranges for each lifecycle selection factor. These assigned values could be tabulated and presented graphically to help visualize and compare each characterization. We postulated that it would be feasible to develop an empirical characterization model for each lifecycle by compiling and analyzing data from a large enough number of projects. To be clear, our goal was to explore possibilities – not compile reams of project data.

Next we began to reflect on the types of methods and guidance that would need to be developed for estimating appropriate characterization data for each project. Presumably, these methods would be related to the above-mentioned empirical characterization data that would be complied for lifecycles. A significant additional challenge would be to develop an effective matching technique that quantifies the degree of project-to-lifecycle fit. We anticipated that some of the methods used in software estimating [4] and COTS selection [5] could be adapted to accomplish this goal.

4 Postulated Lifecycle Selection Process

We postulate that a lifecycle model selection process would flow along these lines:

1. As a pre-condition, that baseline characterization data ("profile") for each lifecycle model would need to be available, likely in a parameterized form, derived from project data collection and analysis.
2. Next, the project objectives, context, assumptions, resources, constraints and priorities would be collected from project stakeholders (e.g. customers and users) and compiled.
3. The list of candidate lifecycles would then be pruned, if possible, to eliminate any obvious incompatibilities between the project attributes and lifecycle attributes.
4. Using the project data, each of the selection factors would be estimated by assigning nominal values to them, possibly by applying adjustment factors (e.g. "multipliers").
5. A matching algorithm comparing the project characterization data with each lifecycle model's characterization profile would be executed to estimate the degree of fit and select the best fit.
6. Sensitivity and trade-off analyses could be performed by varying the characterization data and conducting independent project cost and schedule estimates.

Clearly the development of such a lifecycle selection process would require considerable effort to vet and refine postulated process, compile and analyze project data, and develop suitable guidance and matching methods.

5 Characterization of Each Lifecycle Development Model

In the sections that follow, we have provided our rough-cut characterization data estimates for the principle lifecycles (see [3], [8] and [9]) to illustrate the first step of this approach. These estimates characterizing each lifecycle have been tabulated (see annex) and graphically presented for each lifecycle. The table in the annex contains the estimates and rationale used to subjectively estimate each factor for each lifecycle model. Observe that some of the estimates in the tables are expressed in very broad ranges (e.g. Medium-to-High).

6 Waterfall Development

Waterfall development is relatively sequential - characterized by development phases, major milestones, and specified deliverables reviewed by stakeholders (depicted in Figure 1 and characterized in Figure 2). Typically, looping back to the previous phase to correct problems is permitted (waterfall is almost never purely sequential). The Waterfall Model has been used in its most disciplined form in the aerospace, military, and related application domains. Formal change control procedures are normally mandated to correct problems in earlier phases. The Waterfall Model encourages thorough requirements development and design, and formalizes milestones, documentation and deliverables.

Waterfall is less adaptive than other models to project changes and market demands, and project visibility is primarily achieved through the delivered documentation. This process is typically used for safety and reliability critical systems where quality and lifecycle maintainability are high priorities, the systems tend to be large and complex, and the requirements are fairly well understood - at least in terms of general functionality and scope. Waterfall is applicable to large, mid-sized, and small projects alike.

Figure 1: Waterfall Lifecycle Model [11]

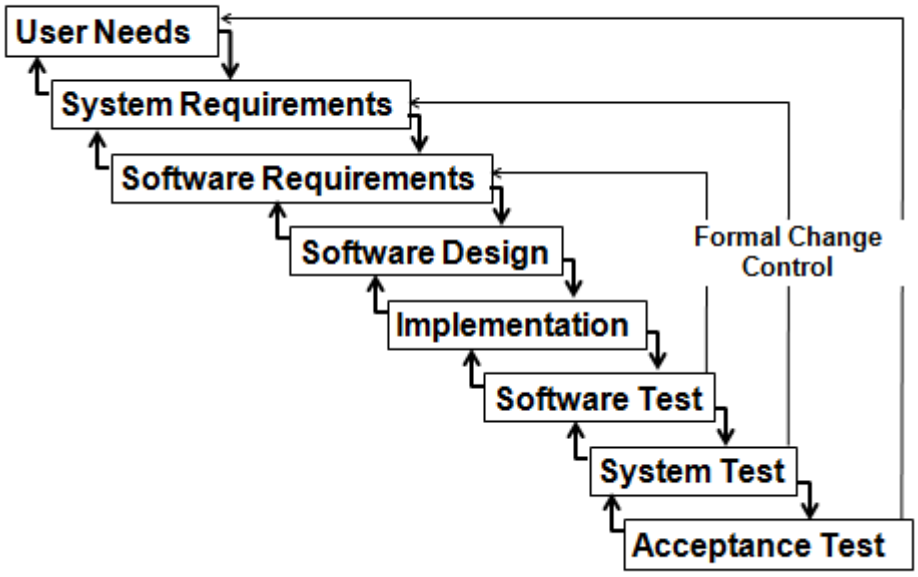
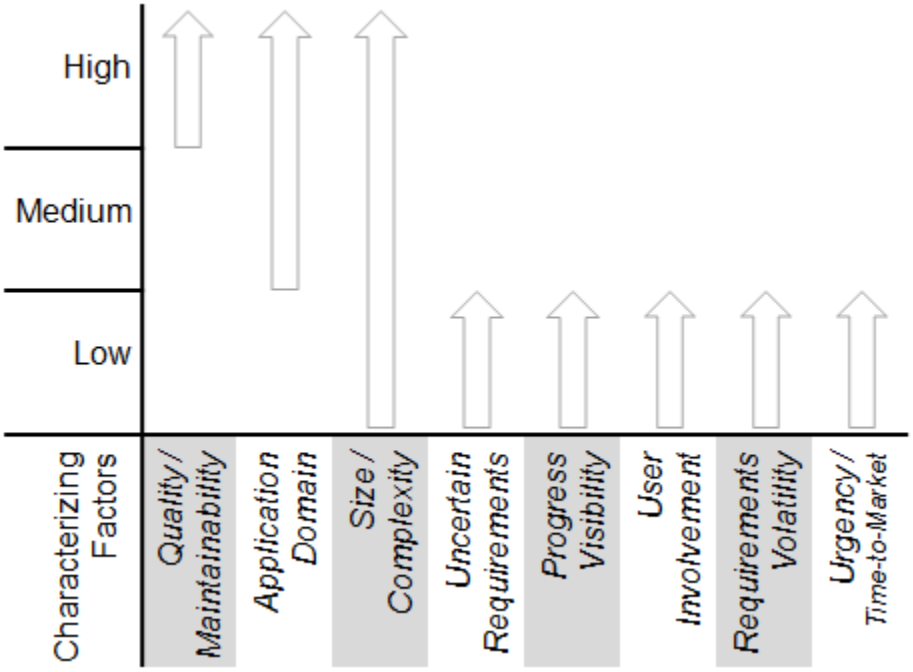


Figure 2: Waterfall Lifecycle Model Characterized



7 Iterative Development

Various iterative models have been developed over the last two decades. They include the incremental, spiral, evolutionary and agile development lifecycles described below. The waterfall model itself has been adapted to be somewhat more iterative than originally interpreted and practiced. The common characteristics of iterative models include: repeated cycles of development, ongoing rework, and parallel (concurrent) development. They generally enable better schedules, early discovery of problems, better customer and user feedback, and visibility into progress by way of demonstrations. However, they can be challenging to manage. The iterative model variants follow.

8 Incremental Development

The traditional view of the Incremental Model is that it is a planned iterative lifecycle that partitions large, complex problems into independent parts, concurrently develops these partitions, and progressively integrates the parts. This should be contrasted with Agile development which has adapted certain, but not all (e.g. up-front requirements and design), aspects of traditional incremental development and is described below. The traditional incremental model is depicted in Figure 3 and characterized in Figure 4.

The traditional Incremental Model first develops a thorough understanding of the requirements, then develops a stable system architecture, and then partitions the system into independent partitions or subsystems for development. Incremental lifecycles are well-suited to concurrent development and both partial and progressive delivery of system capability and have been extensively applied in aerospace and military domains to scale up the waterfall process for very large and highly critical projects. Mapping requirements to increments can be challenging and unanticipated changes to requirements and the architecture can break across increments and imply major rework later on.

Incremental delivery allows for a degree of requirements uncertainty since poorly understood requirements can be pushed off to later increments and product releases. The Incremental Model also

offers positive visibility into progress at each release. Observe, however, that management and technical processes must be able to modularize the project to be able to run such projects without losing control. This implies devoting early effort to baselining as much of the requirements as possible and stabilizing foundation components and the most critical architectural modules as early as possible. In comparison to the waterfall, an incremental lifecycle will increase both technical and management overhead, will yield longer schedules, and increase overall project costs. However, large and very larger projects can be tackled very effectively by way of the incremental lifecycle model.

Figure 3: Incremental Lifecycle Model [11]

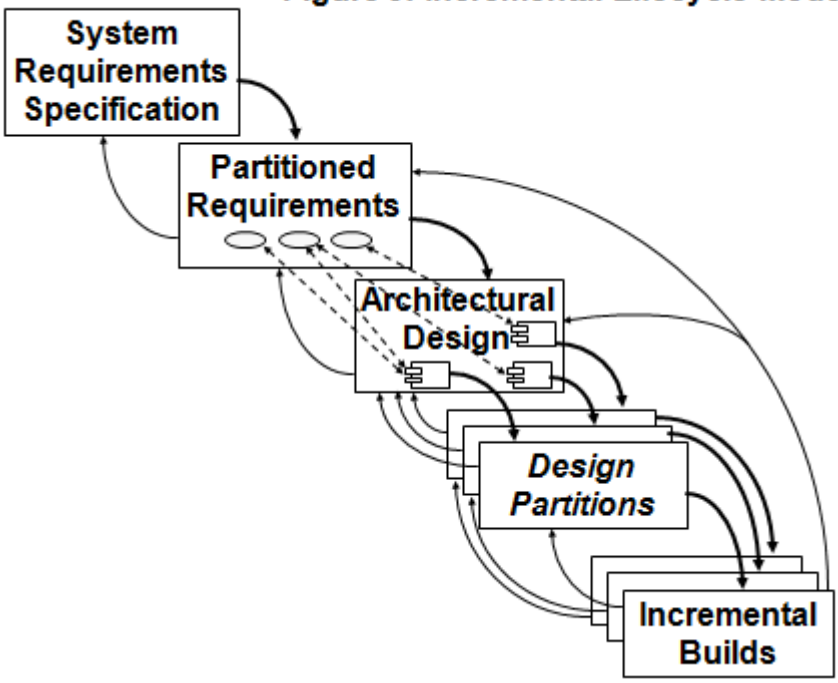
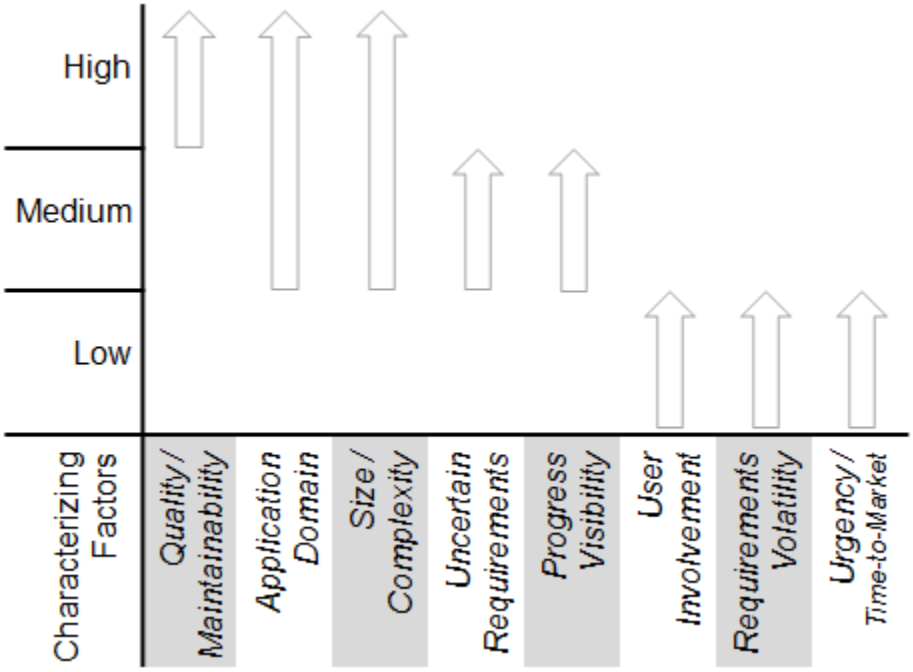


Figure 4: Incremental Lifecycle Model Characterized



9 Spiral Development

Originally proposed by Dr. Barry Boehm [7], the Spiral Model is an innovative process-rich model that possesses iterative and incremental features and can be challenging to interpret. Several newer models have spun-off from Boehm's model. The Rational Unified Process (RUP) [8] is a comprehensive iterative adaptation of this model – subsequently retro-fitted to fit better with Agile development. The various Agile methods [9] are also spin-offs in certain respects.

The Spiral Model, depicted in Figure 5 and characterized in Figure 6, advocates a risk-driven and plan-oriented iterative approach where each spiral is a development iteration that aims to establish a plan for the next spiral (a.k.a. iteration). Risk assessments prior to each spiral determine the activities scheduled for a given spiral/iteration. Reviews at the end of each spiral include an assessment of "lessons learned" that feed the next spiral. Early spirals systematically focus on consolidating the requirements and exploring technical problem areas through prototyping and simulating. Later spirals transition in more waterfall-like iterations of development. Concurrent spirals can represent increments of development.

The spiral lifecycle process model is flexible in that it to accommodate adaptation – the process engineer can tailor as much or as little discipline into the spirals as they may wish. This model encourages, but does not mandate, the notion of continuous risk assessment and the incorporation of both incremental and evolutionary development approaches. The Spiral Model enables many of the capabilities of the incremental and waterfall models while offering better project visibility, incorporating more user involvement, and dealing better with changing requirements. However, the Spiral Model can also be adapted to incorporate much technical and management overhead and therefore has the potential of escalating project schedules and costs beyond that of the waterfall and other iterative models. Project management and contracting can become more challenging than some of the other models as the Spiral Model can be adapted to specify disciplined processes for controlling requirements, architecture, risk mitigation, and process improvement (lessons learned) activities.

Figure 5: Boehm's Spiral Model [11]

Adapted from [7] (considerably simplified)

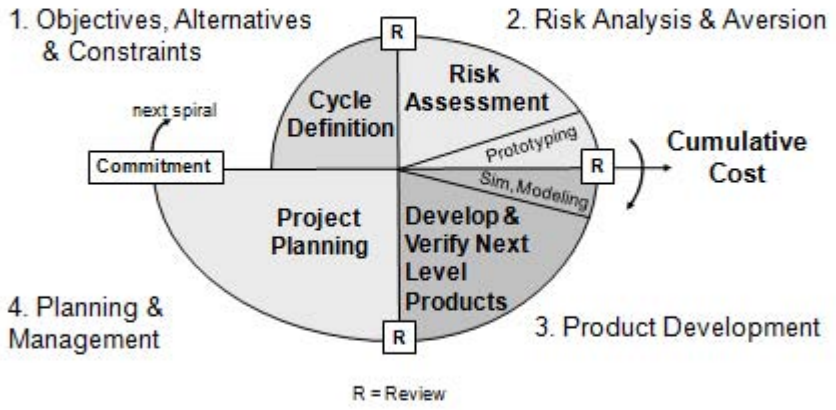
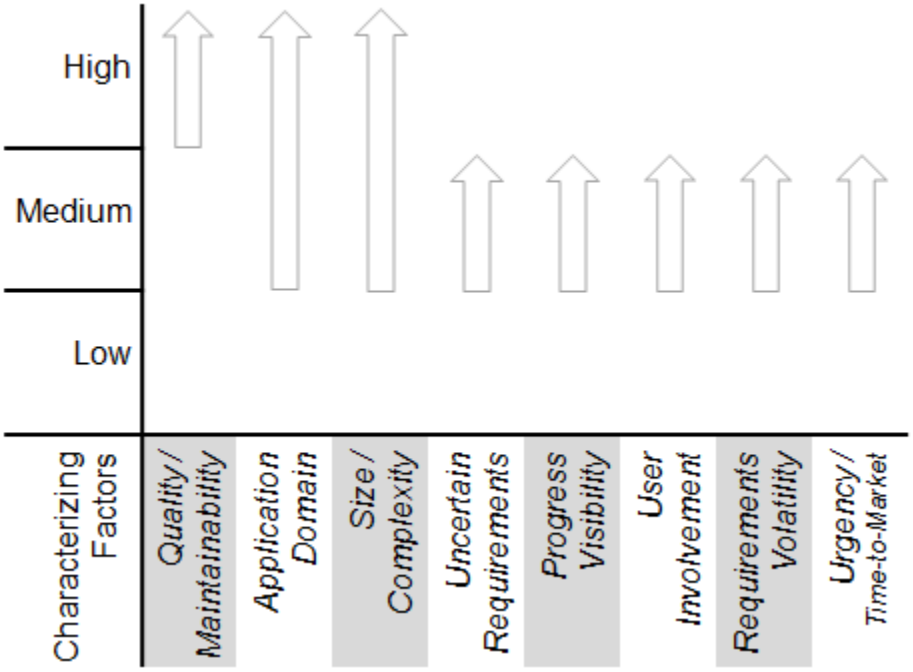


Figure 6: Spiral Lifecycle Model Characterized



10 Evolutionary Development

The Evolutionary Model is an iterative exploratory development model for solving hard (non-trivial) technical problems and uncertainties. Work products of this model are designed to discover technical solutions and elicit feedback. The Evolutionary Model focuses project stakeholders (developers, managers, customers, and users) on feasibility and requirements rather than on the targeted solution. Detailed functions and features, as well as product qualification tasks such as reviews and testing can be avoided. However, there is a danger that managers and customers may assume that the prototypes they are shown are of deliverable quality – when they are typically not - which can unreasonably inflate expectations. The Evolutionary Model is depicted in Figure 7 and characterized in Figure 8.

Evolutionary lifecycles aim at providing good visibility into the various technical problems encountered as well as encouraging user involvement and customer feedback. However, it is not intended for maintainability or the delivery of operational releases. Rather, this model should be used to explore difficult/risky technical issues and/or to validate uncertain requirements (e.g. user interface requirements, heuristic computations, and fuzzy decision logic). The Evolutionary Model is also very useful in experimental or research settings where the products are for more personal use, for example to produce a proof-of-concept prototype. The work products are not considered production quality, at least for large, medium-sized, complex or mission-critical projects where lifecycle maintainability is a priority. However, evolutionary development is a very useful process for front-ending a more structured waterfall or incremental development process. In fact, iterative models like the Rational Unified Process (RUP) for software development incorporate evolutionary development - but they endeavor to ensure that their work-products go through adequate design, review and testing steps before release into test and production.

Figure 7: Evolutionary Lifecycle Model [11]

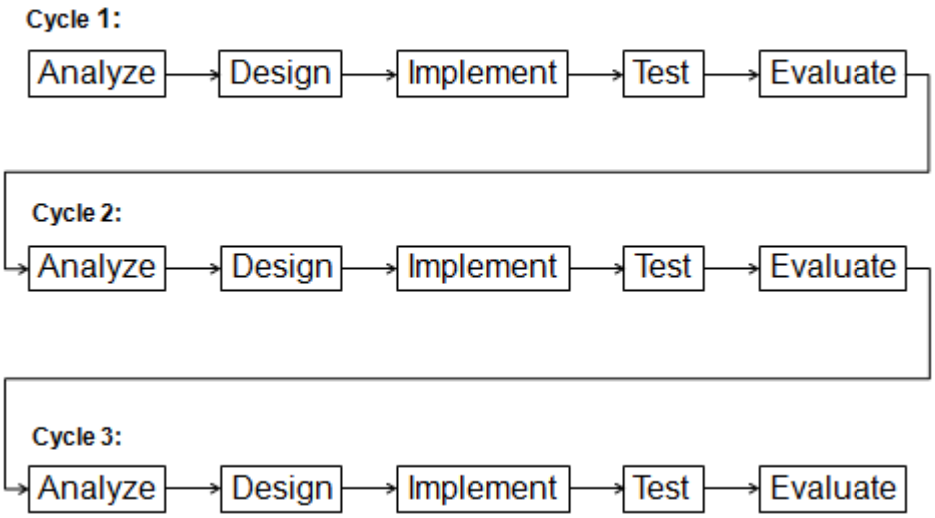
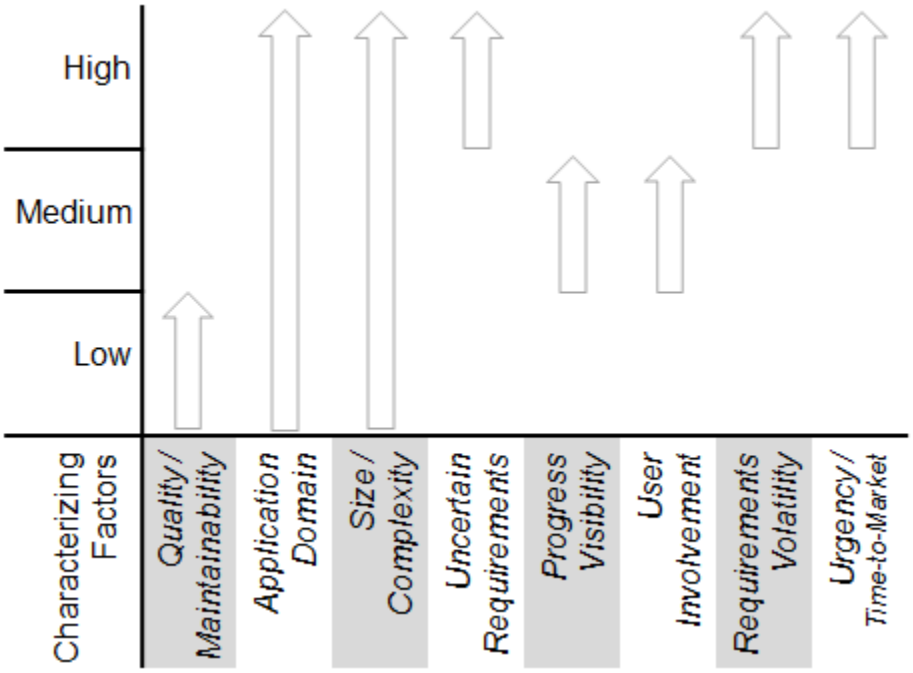


Figure 8: Evolutionary Lifecycle Model Characterized



11 Agile Development

Agile development represents a family of lighter weigh processes (e.g. XP, Scrum, etc.) that have been popularized over the last dozen years. Their roots can be traced back to iterative, evolutionary, and incremental development lifecycles which explains why agile is said to be iterative and incremental. These methods focus on developing working software over documentation, and they are said to embrace change and close customer involvement. Solutions are built up from customer and user-provided "stories" over short development iterations (typically 1-4 weeks). Stories are prioritized and put into a backlog; planning is "time-boxed"; some methods advocate pair-programming (e.g. XP) and/or "test-driven development" (TDD). Agile development is depicted in Figure 9 and characterized in Figure 10.

Like evolutionary development, these models deal with uncertain requirements – typically by requiring intimate customer and user involvement. Agile processes such as Scrum nudge evolving work products towards delivery and release by way of constant testing and re-factoring. The emphasis on working code contributes to their ability to demonstrate evolving functionality while light-weight documentation facilitates adaptability to changing circumstances. Favoring on-going re-factoring as stories are integrated, over big up-front requirements analysis and architectural design, enables early visibility and feedback. For large complex systems such continuous re-factoring could lead to fragile solutions and maintainability problems, especially if documentation is lacking. Also, customers may not participate as promised, and close customer involvement across large-scale projects tends to be more difficult to manage. Given the reduced emphasis on documentation, Agile projects may be more highly dependent on the skill and memory of their developers which may expose projects and companies to staff turnover risk. Overall, agile development appears to be more suitable for small-to-medium-sized projects of moderate criticality where schedule pressure and quickness to market are high priorities.

Figure 9: The Life of an Agile Story [11]

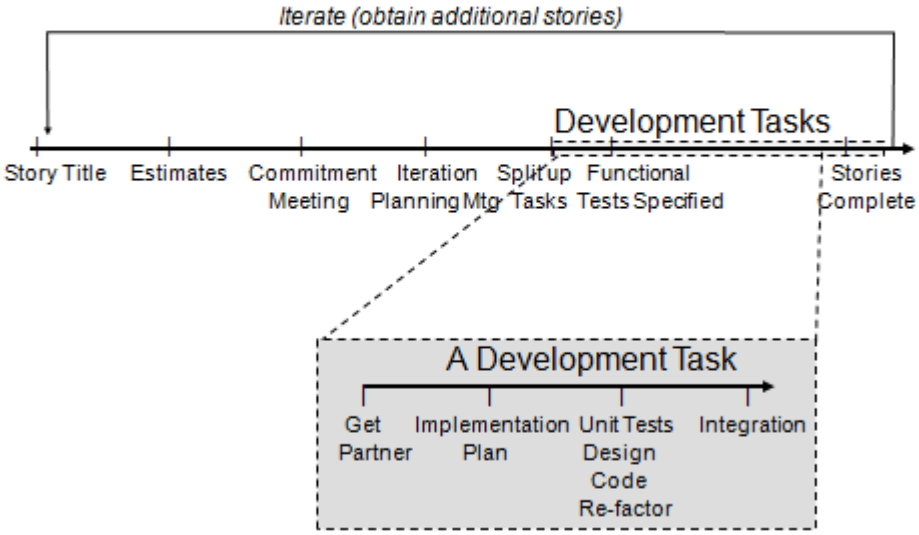
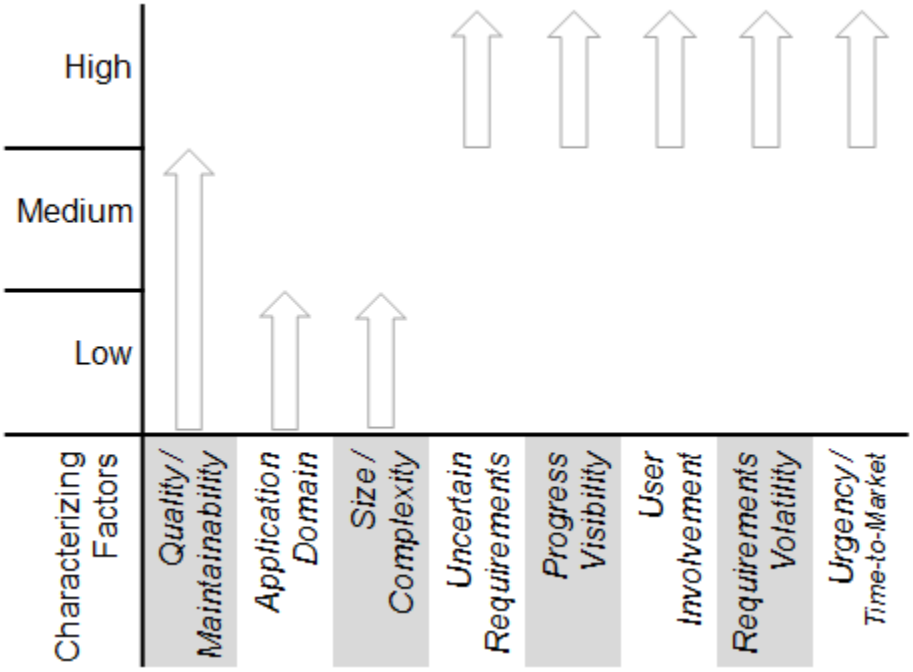


Figure 10: Agile Lifecycle Model Characterized



12 Project-to-Lifecycle Matching

Although we have not explored the problem of matching projects to lifecycles, we have attempted to illustrate the idea in the figures below.

Figure 11 presents a visualization of the project-to-lifecycle match for a hypothetical project that has already been characterized. The characterized project is represented as a single line graph that is superimposed over each of the (five) already characterized lifecycles. Figure 11 illustrates, for example, that the Waterfall Model meets or exceeds the project's needs in 5 out of 8 areas – failing to meet needs in 3 areas. Meanwhile, Agile development appears to meet or exceed project needs in 6 out of 8 areas, and the Spiral model appears to satisfy all eight factors. In search of other lifecycle choices, the matching process could go on to quantify the degree to which each factor is met, exceeded, or fails to be met by each lifecycle; and adjustments could be explored by reassessing or changing the character of the project, and/or tailoring/adapting each lifecycle to better fit project needs.

Meanwhile, Figure 12 illustrates a hypothetical project whose characteristics fail to match any of the discussed lifecycles. Observe in particular that this project aims to deliver a highly maintainable product, expects that the requirements will be very uncertain, and acknowledges that there will be considerable time-to-market pressure. This figure nicely verifies our intuition that none of the lifecycles appear to be capable of delivering such a highly constrained project. These mismatches between the project's apparent needs and what the various lifecycles best support imply that the project needs as articulated are unrealistic or unachievable. Probably the project's needs should be re-assessed. A potentially risky alternative to consider would be to put in place a new or hybrid lifecycle model that somehow overcomes the demands of the project.

Figure 11: Illustrating a Project Matched to Lifecycles

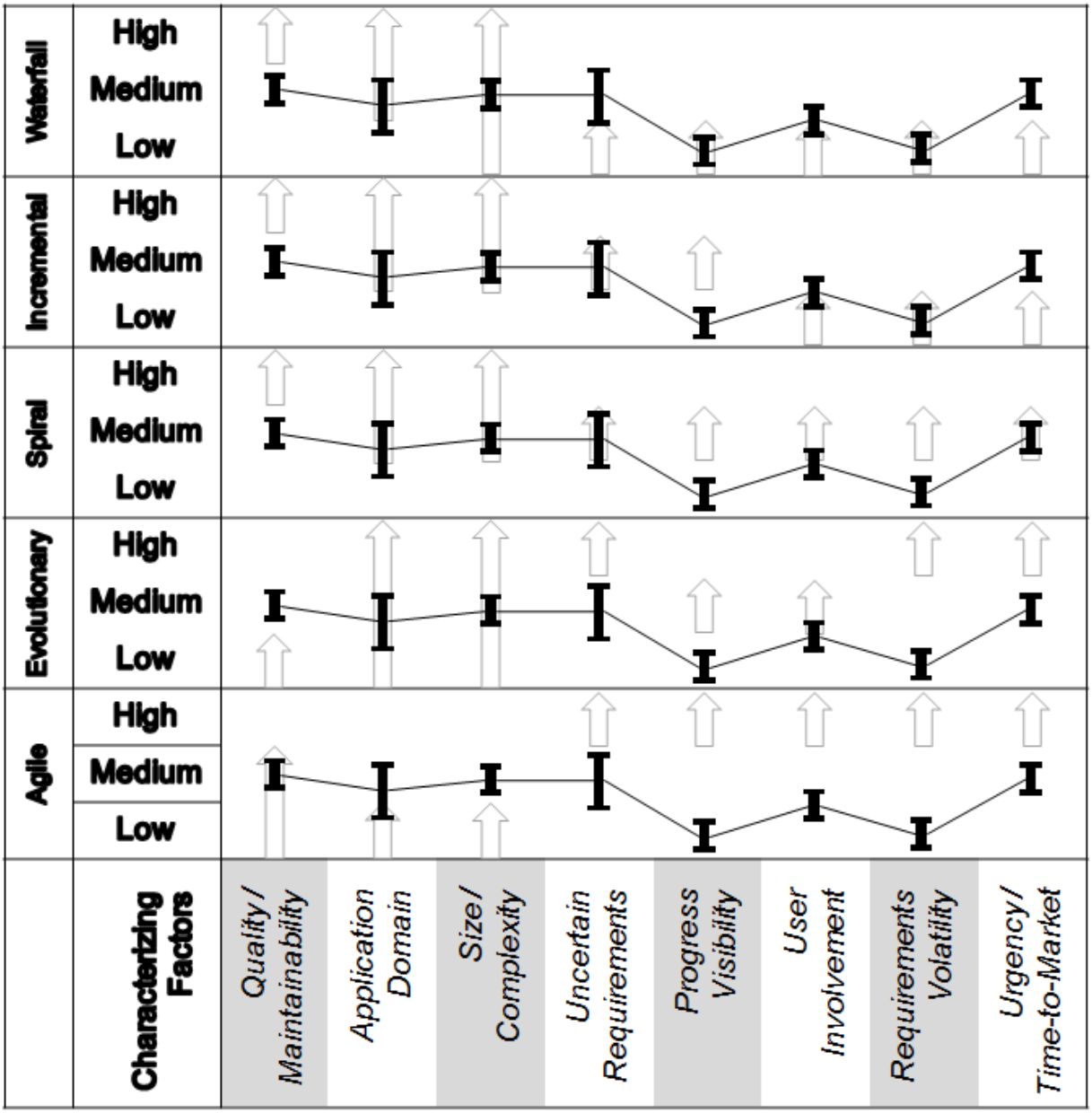
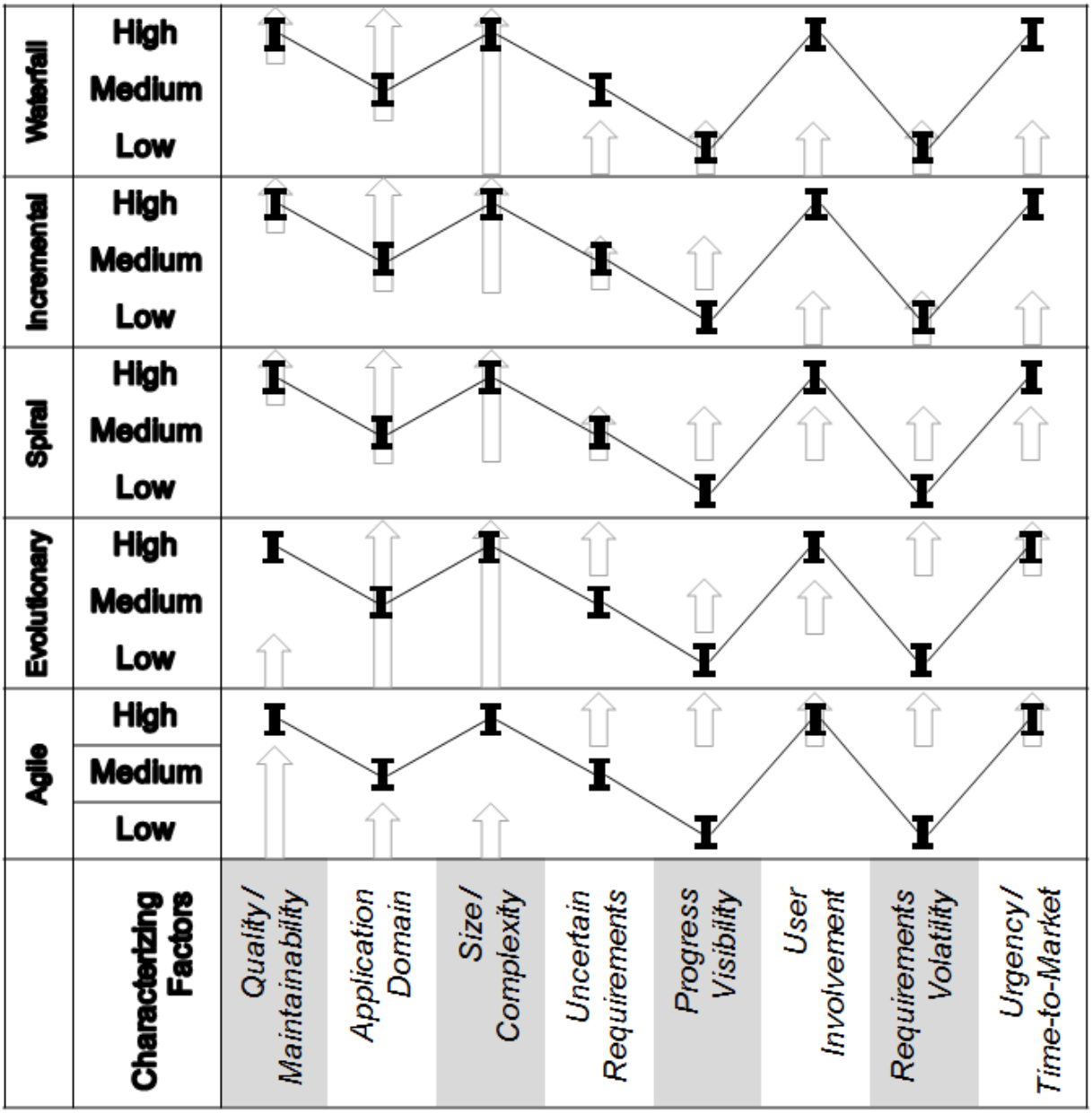


Figure 12: Illustrating a Project that Fails to Match a Lifecycle



13 Summing Up

This paper has suggested a general approach for lifecycle model selection and has explored the first step of such an approach, namely, to characterize the most popular lifecycle models in terms of eight critical selection factors. We acknowledge that the suggested characterization values have been subjectively arrived. Nevertheless, they have been distilled from the authors' 50 years of collective industry experience estimating, planning, managing and supporting systems and software projects. We therefore standby our assertion that the strategy put forth to characterize lifecycle models by way of critical project characterization factors could lead to a practical, balanced approach for selecting a suitable lifecycle model for a given project. We invite constructive criticism and validation from readers to help move towards such a goal.

An empirical approach for determining these factors and making trade-offs when selecting a lifecycle model has not been presented. However, it is hoped that this paper will motivate others to mount research and assessment projects that will build on the ideas presented.

References

[1] Kal Toth, "Which is the Right Software Process for your Problem?",Software Assoc of Oregon, 2005

[2] Kal Toth, "What's the hard part of software development anyway?",Software Assoc of Oregon, 2007

[3] Barry Boehm, Richard Turner, "Balancing Agility and Discipline", Addison Wesley, 2004

[4] Kal Toth, "Selecting Software Estimating Techniques that Fit the Software Process", PNSQC, 2008

[5] Kontio, A Case Study in Applying a Systematic Method for COTS selection, 1996, ICSE

[6] Dan Brook, Kal Toth, "Levels of Process Ceremony for Software Configuration Mgt", PNSQC, 2007

[7] Barry Boehm, "A Spiral Model for Software Development and Enhancement", Computer, V21,1988

[8] Philippe Kruchten, "The Rational Unified Process: An Introduction", Addison Wesley, 1999

[9] Alistair Cockburn, "Agile Software Development", Addison Wesley, 2002

[10] Kal Toth, "Change in Development Methodology", Hughes Systems Division Report, June 1, 1993

[11] Kal Toth, "Principles of Software Engineering", Lecture Notes, 2005-2010

Annex: Characterized Lifecycle Development Models

Waterfall Model		Relatively sequential with development phases, major milestones, & specified deliverables reviewed by stakeholders
Quality/ Maintainability	H	Advocates thorough requirements, architecture & design specification which facilitates quality reviews and inspections, and fosters quality documentation for downstream maintenance
Application Domain	M, H	Waterfall is best suited for projects that have stringent quality/criticality requirements, that is, those with high reliability, maintainability, availability, safety, and/or security requirements (are “mission-critical”)
Size / Complexity	L, M, H	A common misconception is that waterfall is best suited for large complex projects. It is more correct to say that waterfall is more suited to mission-critical projects of virtually any size
Uncertain Requirements	L	Waterfall assumes the requirements are fairly well understood
Progress Visibility	L	High levels of process formality and document deliverables will satisfy contractual requirements and facilitate progress payments but may give a false impression of true progress being achieved
User Involvement	L	User involvement typically confined to requirements phase
Requirements Volatility	L	Not very responsive to changing requirements due to formal change control process
Urgency	L	Although some iteration is allowed, the planned/controlled character of the waterfall model renders it less responsive to market forces.

Incremental Model		An iterative process that partitions large complex problems into independent parts, some of which may be mission-critical, concurrently develops and integrates the parts, and delivers multiple releases
Quality/ Maintainability	H	Advocates thorough requirements, architecture & design specification which facilitates quality reviews and inspections, and fosters quality documentation for downstream maintenance
Application Domain	M, H	Incremental development is well-suited for projects where mission-critical partitions can be identified and allocated to independent development increments – this is not applicable to small projects
Size / Complexity	M, H	Incremental development is well-suited for larger and more complex projects – project partitioning not needed on smaller projects
Uncertain Requirements	M	If requirements are well-partitioned at the top level, incremental development can localize the problem of uncertain requirements.
Progress Visibility	M	Concurrent development enables independent monitoring of parts which can elevate visibility into project progress
User Involvement	L	User involvement typically confined to requirements phase
Requirements Volatility	L	Not very responsive to changing requirements due to formal change control process
Urgency	L	The planned/controlled character of the incremental model renders it less responsive to market forces.

Spiral Model		A risk-driven plan-oriented iterative model where each spiral is a development iteration that aims to establish a plan for the next spiral (a.k.a. iteration).
Quality/ Maintainability	H	Risk-driven approach with lessons-learned well-suited systematically developing quality specifications for life-cycle maintenance.
Application Domain	M, H	The risk assessment aspect of this progressive development approach is well-suited to address the technical and management risks posed by mission-critical requirements
Size / Complexity	M, H	The risk assessment aspect combined with process decomposition into development spirals and lessons-learned is well-suited to address the technical and management risks posed by large complex applications
Uncertain Requirements	M	Early spirals systematically focus on consolidating the requirements & exploring technical problems thru prototyping & simulation
Progress Visibility	M	Though more challenging for project management, the risk-driven approach provides progress visibility from an additional perspective (in addition to delivered documents and functionality)
User Involvement	M	Though user involvement is not explicitly call up, a majority of technical and management risks encountered will have direct impacts on the customer and/or users which motivates planning and execution explicitly incorporating customer and user involvement
Requirements Volatility	M	The spiral model represents a compromise solution for dealing with requirements volatility. Risk assessment will help stabilize requirements early in the project in the areas that are most changable
Urgency	M	The risk-driven strategy can be employed to identify spirals of development that should be accelerated and developed independently from challenging areas to meet urgency and time-to-market needs

Evolutionary Model		An iterative exploratory development model for solving hard (non-trivial) technical problems and uncertainties – they are not considered to be of operational/deliverable quality
Quality/ Maintainability	L	Evolutionary development does not deliver solutions of deliverable quality – however, this strategy may have a positive impact on the quality of final deliverables and is likely to improve schedules and costs
Application Domain	L, M, H	Evolutionary development is generally applicable across all domains
Size / Complexity	L, M, H	Generally applicable regardless of project size or complexity. Larger projects are will offer more opportunities for evolutionary development; very small projects may not need more that evolutionary development
Uncertain Requirements	H	A key application of evolutionary development is to explore requirements so as to reduce uncertainty, schedules and costs by reducing the need for supporting requirements change
Progress Visibility	M	Evolutionary development encourages focusing on project uncertainties and will improve visibility into progress rather than hiding uncertainties until “just too late”
User Involvement	M	Focusing on usage and user interface issues, especially early in the project, will encourage user and customer involvement and reduce schedule and cost risks
Requirements Volatility	H	Early exploration into requirements with the involvement of users can help stabilize requirements early and reduce requirements churn
Urgency	H	This exploratory approach can be used to focus attention on difficult problems early thereby removing risk from the project’s critical path

Agile Model		Focus is on developing working software over documentation, embracing change and close customer involvement. Stories are prioritized and put into a backlog; planning is "time-boxed"; some methods advocate pair-programming (e.g. XP) and/or "test-driven development" (TDD).
Quality/ Maintainability	L-M	Agile solutions favor frequent re-factoring over developing a stable architecture which may result in brittle operational solutions Emphasis on working software over documentation makes agile development more vulnerable to employee turnover
Application Domain	L	Although various authors have claimed that Agile development is suitable for mission-critical applications, documented evidence is anecdotal at this time
Size / Complexity	L	Although various authors have claimed that Agile development is suitable for large and complex applications, documented evidence is anecdotal at this time
Uncertain Requirements	H	Agile is specifically oriented towards addressing requirements uncertainty - user stories are described very informally, presented as prototypes to users, and refined until accepted by users
Progress Visibility	H	Progress visibility is realized in terms of the released stories and their functions and features. Note that completeness of testing and documentation are not explicitly used to provide progress visibility
User Involvement	H	Agile is heavily dependent on customer buy-in and commitment to ensure that users are involved deeply in the project
Requirements Volatility	H	All stakeholders are committed to embrace change and prepared to re-engineer and re-factor the current software build at any time
Urgency	H	Each story is meant to be releasable to the user base – hence, provided the latest build(s) is of releasable quality, an operational product should be available on a timely basis

Summary Characterization of Lifecycle Models	**Waterfall**	**Incremental**	**Spiral**	**Evolutionary**	**Agile**
Quality/ Maintainability	H	H	H	L	L-M
Application Domain	M, H	M, H	M, H	L, M, H	L
Size / Complexity	L, M, H	M, H	M, H	L, M, H	L
Uncertain Requirements	L	M	M	H	H
Progress Visibility	L	M	M	M	H
User Involvement	L	L	M	M	H
Requirements Volatility	L	L	M	H	H
Urgency	L	L	M	H	H

Reduce QA Cost by Improving Productivity & Test Optimization

Author(s)
Rajesh Kumar (rajesh_kumar_5@uhc.com) and Ajay K Chhokra (ajay_k_chhokra@uhc.com)

UnitedHealth Group Information Systems, Unitech Cyber Park, Tower C ,Floors 1-4
Haryana, 122001, India

Abstract

"Reducing QA Cost" is the buzzword in IT industry. Testing Costs can be reduced by two ways – by using Test Optimization techniques, and by increasing productivity. Test Optimization focuses on efficient Test methodologies, and reducing redundancy and waste, by developing and executing optimized Test suite. Test productivity focuses on increasing the productivity of the Testing process. In this paper, we are doing a deep dive on Testing Optimization & Productivity Improvement techniques and their impact on QA cost reduction.

To reduce QA Cost, an organization needs to establish the QA cost baseline and processes to capture and track the QA cost on regular basis. Organizations also should establish the goals to reduce the same on a year to year basis. In this paper, we have shown effective methods which help the organization to reduce QA costs by implementing various Test Optimization techniques and Productivity Improvement techniques.

Biography

Ajay K Chhokra *- Ajay is part of UnitedHealth Group, India IT Leadership Team and is responsible for driving Testing Competency and for establishing Global Testing Center of Excellence in Global Sourcing and Delivery center. His initiatives around Delivery Excellence, Maturity Assessment, Test Governance and Test Regression Automation of Legacy Systems have brought about productivity, predictability and Quality improvements in UnitedHealth Group IT deliveries. Ajay has been a Key-note speaker and presenter in many national and international Testing Forums. He has authored a book titled "Excellence in Software Quality". During his rich and diverse professional career span of 25 years, he has gained vast experience in Quality Assurance, Process Improvements and Testing across leading product and service companies of the IT Industry.*

Rajesh Kumar - *is working as a Project Manager in United Health Group IT and is responsible for Project delivery of Web Portals department, driving Structured Quality Improvement Plan, and Reducing QA cost initiatives. He has extensive experience in Web Portals Testing, Regression Optimization, Test Automation, Test Optimization techniques implementations include OATS, RBT, DTT etc, Test Automation of Mobile and Cross Browser Testing. His initiatives around reducing QA cost, Improving Productivity, Resource Optimization, and Test Optimization techniques implementations has helped a lot for Quality improvements in UnitedHealth Group IT deliveries.*

1 How to Capture QA Cost

There are various ways to capture QA cost in an enterprise. Following are the most frequently used parameters for capturing QA Cost:

(1) Project Hrs (All Roles) charged to the Testing phase – Cost associated with different teams like Development, System Analyst, QA team and others charging to Testing phase

(2) **Project Hrs (QA Role) charged to "Integration and Testing (I&T) phase" –** Cost associated with QA role charging to I&T phase only

(3) **Project Hrs (QA Role) charged to Software Development Life Cycle (SDLC) phases –** Cost associated with team members performing the QA role charging to different phases of SDLC like System Analysis, Construction, Testing etc.

(4) **Project Hrs(QA Roles) Charged to different line Items within "Integration & Testing" phases, –** Cost associated with QA role charging to different sub-parts of I&T phase

Software Industry Flavor

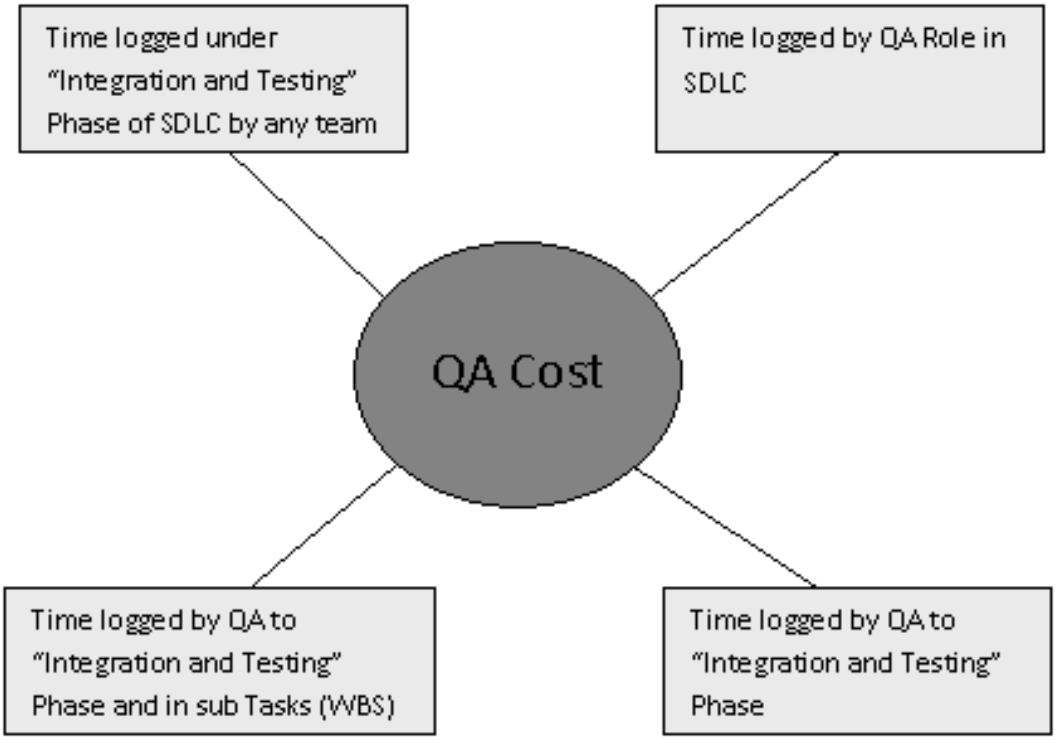


Figure 1: QA Cost Time Tracking

2 Tracking QA Cost

To improve anything, one needs to establish the baseline. Similarly, to track QA Cost, one needs to establish the baseline and then publish the QA cost on regular intervals (could be monthly or quarterly). Tracking QA cost on a regular basis is a tedious task; one needs to capture all the charges (*hrs*.) information in different phases of SDLC.

Like any other organization, we also have a Time tracking system to captures the efforts spent by team members on various project activities. We have taken the data out of our time management system (Plan view) and plotted the Month-to Month effort for the given project. The Figure 2 graph shows:

- Monthly QA cost
- # of associates playing QA role working for the project
- Baseline cost of previous Year
- Total SDLC Hours

The below mentioned practice can be used to track the QA cost:

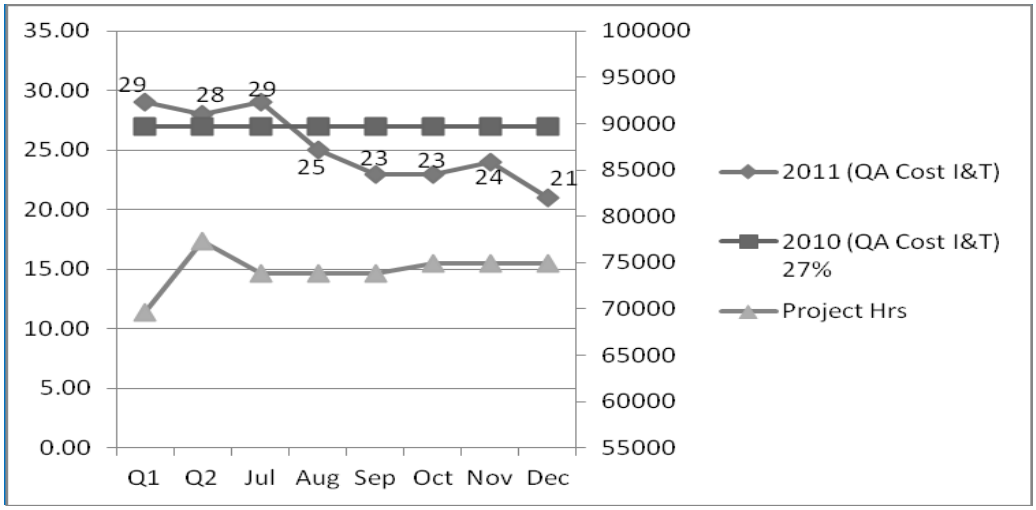


Figure 2: QA Cost & Project Hrs

We also have taken the Defect data from Defect Management System. By plotting both Project Effort data and Defect data, one can draw analysis of “Impact of QA Cost on Overall Quality” and/or Impact of Productivity Improvement on overall QA Cost. The Figure 3 shows the depiction of Cost Vs Quality.

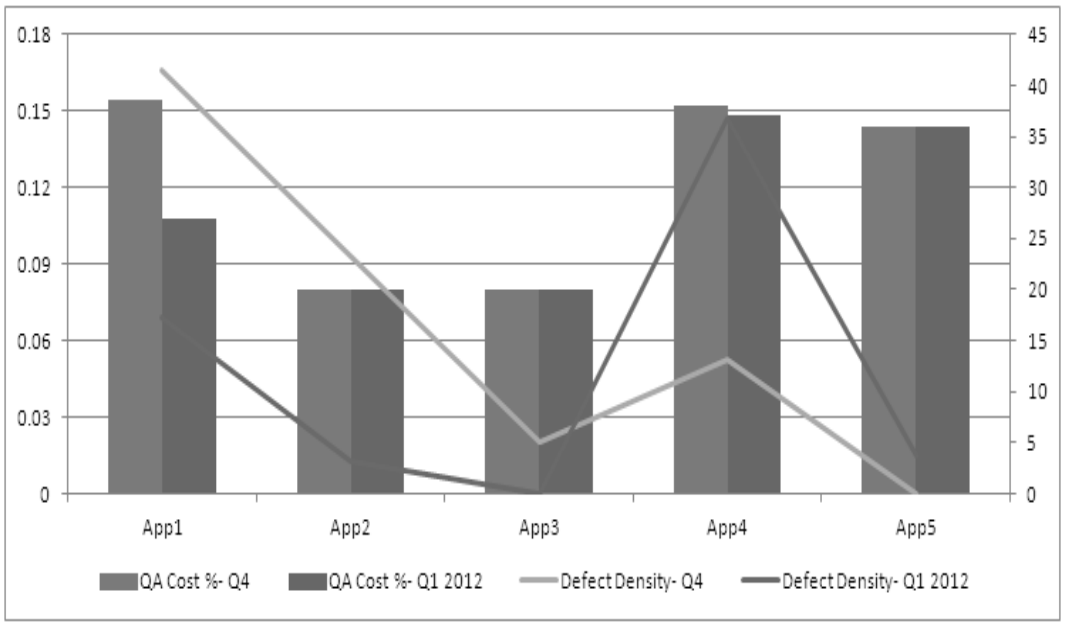


Figure 3: Cost & Quality

3 Test Optimization Techniques

In the real world, Business expects test teams to do thorough and exhaustive testing, but project cost and timeline does not make it feasible to test all possible variation and combinations of all possible test scenarios. Test Optimization techniques helps in adjusting a testing process so as to optimize some specific set of parameters without violating some constraint. The goal of Test Optimization techniques are:

- Minimizing Cost

  Minimizing number of Test cases, which helps in:
  - Reduction of Test Design,
  - Reduction of Test Case and executing time,
  - Reduction of Testing effort
- Maximizing Throughput and/or efficiency
  - Maximum Test coverage with minimum Test Cases,
  - Maximum throughput with efficient Test Cases
  - Minimizing time to market

With minimum Test Cases and maximum coverage, overall Testing get completed in shorter span of time, thus contribute in overall goal of minimizing time to market.

Different Test Optimization Techniques

There are many Test Optimization techniques available in Industry. The most popular are

- Orthogonal Array Test Strategy (OATS)
- Regression Optimization
- Automation
- Risk Based Testing (RBT)

3.1 OATS

The Orthogonal Array Testing Strategy (OATS) is a systematic, statistical way of testing pair-wise interactions. It provides representative (uniformly distributed) coverage of all variable pair combinations. This makes the technique useful for testing of software components. Orthogonal Array is a technique which will help minimize the test cases to an optimum number while maintaining an adequate level of test coverage.

Case Study I:
We have implemented OATS on one of the Web Portal projects having 5000 QA hrs.. This was a big and complex project to test all possible combinations with an expectation to finish it within 3 months with a team size of not more than 8 testers and ensuring good coverage.

Action Plan:
As there were too many combinations on the portal application, team decided to use OATS. A feasibility analysis was done on the same project and came out that OATS is feasible on 40% of the project requirements and it can save us lot of effort and even enhance the functional coverage as well in comparison to manual written scenarios.

Results:
Due to implementation of OATS, we are able to save the below shown effort in Figure 4 on the same project:

- Testcases number Reduction = 40%
- Test Design effort Saved = 480 Hrs
- Test Execution Effort Saved = 600 Hrs

Total effort saved = 1080 hrs
Total Effort reduction: 21.6%

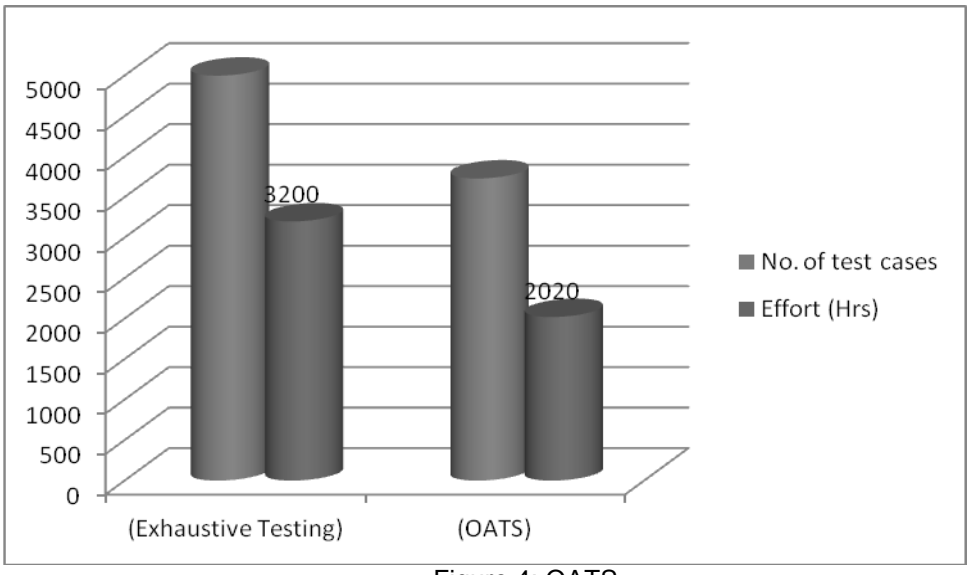


Figure 4: OATS

3.2 Regression Optimization

With every release, the regression Test suite gets bigger and often has many redundant and obsolete Test scenarios. It is a challenge for the Regression team to reduce both size (in terms of scripts) and cycle time, and keep the cost of regression validation low. If a proper strategy is not in place, the growth and size of the regression suite cannot be stopped and ultimately it will be less effective, very time consuming and more effort would be required to do regression testing. The case study given below demonstrates how regression optimization helps to reduce the number of test cases, effort reduction and regression cycle time reduction. Figure 5 shows the Regression Optimization Process:

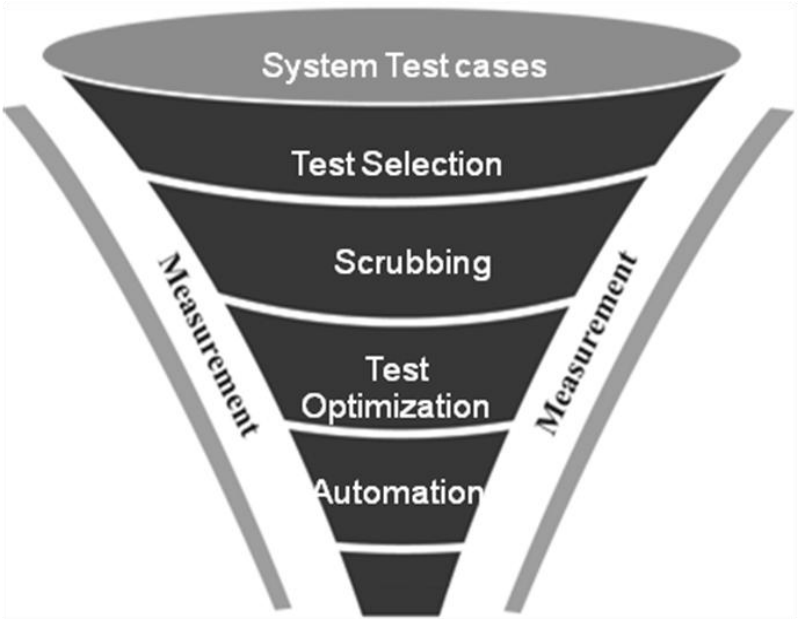


Figure 5: Regression Optimization Process

Key Benefits:

- Effective Test Selection Criteria
- Remove Redundancy
- Improved Quality
- Low Maintenance
- Effort and Cycle Time Reduction
- Increased Functional Coverage
- Reduced analysis effort of failed scripts
- Controlling the growth of regression suite
- Less Testdata Issues

Case Study

One of our portals applications had 3200 Regression Automation Scripts, and those scripts needed to be executed and maintained in each quarterly release. Three testers were working for

2 weeks to execute one regression cycle i.e. 240 Hrs. Two cycles need to run for each release, for a total of 480 hours each release..

Goals:

- Reduce the regression cycle time by 50%
- Reduce QA cost by 50%
- Increase the functional Coverage
- Remove the redundant scripts
- Run More number of regression cycles in one release
- Migration of Regression suite from QTP 6.5 to QTP 9.0

Actions Taken:

- Decided to Re-design the regression suite in two Phases
- Framework Architecture changed, data driven framework implemented
- All the scripts analyzed and merged the same modules scripts with each other.
- Brainstorm sessions arranged to analyze the existing scripts and created new testcases.
- Number of scripts steps increased from 6 steps to 20 steps.
- Removed the redundant scripts
- New functionalities added to the scripts and new Functions to the framework
- Effective Reporting module implemented.
- Virtual Environment is used for test execution
- Scripts Migrated from QTP 6.5 to QTP 9.0

Results:

- Number of Scripts reduced from 3200 to 150. **95% reduction in the number of scripts.**
- Automation penetration increased from 70% to 85%.
- **Regression execution effort reduced by 240 Hrs per cycle to 80 Hrs per cycle. 66% effort reduction**.
- Able to run more number of regression cycles.
- **Functional coverage increased from 75% to 90%.**
- **Defect identification increased by 20%**
- Efficient & Intelligent scripts
- Scripts Migrated from QTP 6.5 to QTP 9.0. Now in QTP11

Details	Time per Script (in Mins) (execution + Analysis)	Total # of Units	TOTAL Effort (Hrs)
- old scripts	7.8	3200	240
New scripts	Execution time= 12 mins Analysis of failed script= 30 Mins *(Assumption 40% will fail) Testdata Identification: 20 Hrs	150	150*12=30 Hrs (Execution time) 150*30=30 Hrs (Analysis time) Re-run:=20 Hrs Testdata Identification: 20 Hrs **Total = 100.00 Hrs**
Total Saving (Per execution Cycle)			**140Hrs**
Total Saving (Per Release)			**280Hrs**
Total Saving (Per Annum)	*Assumption 5 Releases		**1400 Hrs**

Figure 6: Regression Optimization Results

3.3 Automation

Automation is the key solution to reduce the effort towards repetitive tasks. With a web application, it is possible to get up to 90% of functional coverage with automation.

One has to track the Automation penetration on a regular basis and set the improvement goals every year. That helps us in reducing the manual effort every year on the repetitive tasks. Regression is the main area where Automation is very effective. Cross Browser testing and Mobile testing via automation also are effective areas to reduce the manual QA effort. Figure 7 shows the Automation Penetration tracking month wise.

Benefits:

- Cost Reduction
- Increased Test Cycles
- Increased Test Coverage – Automation
- Cross Browser Testing
- Mobile Solutions

Timeframe	2011- Baseline	2012 Q1	April	May	2012 Q3 - Projections	2012 Q4 - Projections
Automation Penetration	31%	38%	39%	47%	55%	60%
Total No. Manual Scripts(to be automated)	9315	9403	9452	9200		
No. of Automation Scripts	2844	3614	3662	4301		

Figure 7: Automation Penetration Tracking

Case Study – Automation Penetration

In one of my areas, Automation penetration for regression testing was 45% at the start of year and we were spending almost 250 Hrs per regression cycle. We decided to increase the automation penetration to 90% keeping in mind that we have to do lot of regression testing for Websphere Product Lifecycle Management upgrade. We worked on that and increased the penetration level to 90%.

After that we analysed our estimation again for regression test execution and decided to reduce the effort to 150 Hrs and hence reduce 100 Hr per regression cycle i.e. 40% reduction effort.

This is not only the one case study, we have seen reduction in QA cost wherever automation penetration is more in regression execution.

We are tracking the Automation penetration on regular basis and set the improvement goals every year, below Figure 8 shows the trend of automation penetration month wise:

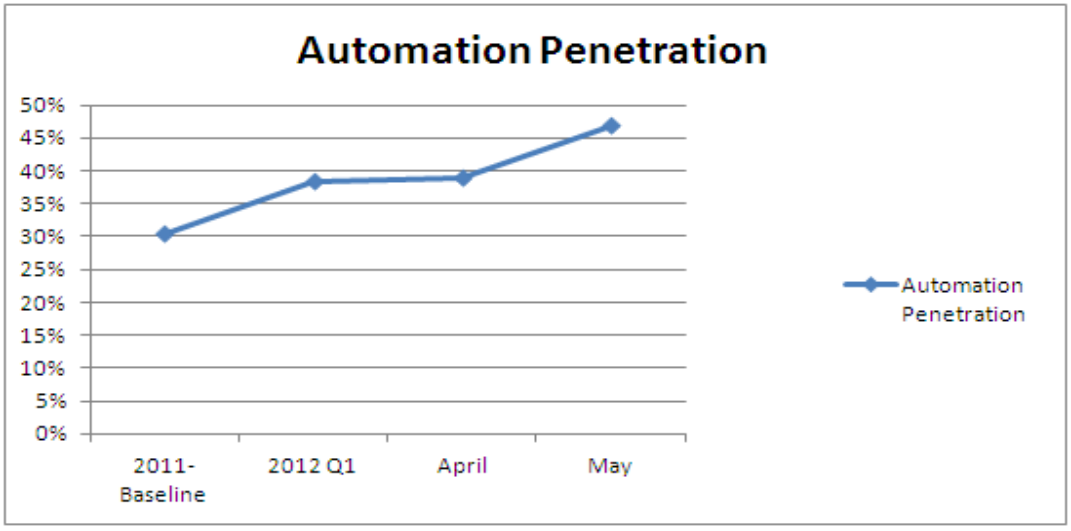


Figure 8: Automation Penetration Trend

The Figure 9 shows another way of tracking automation penetration year by year basis and team can set improvement goals:

					Beseline	Target	Actual	
Application	Automation	2008	2009	2010	2011	2012	2012 0 Actual	No. of Automation Scripts
App1	Regression Execution - Hours spent /Cycle	120	120	80	70	50	50	
	% Coverage - Production Checkout	NA	100%	100%	100%	100%	100%	24
	% Coverage - Regression suite	80%	90%	90%	90%	90%	90%	422*
	Cross Browser Testing	0%	0%	0%	0%	80%	80%	20*
App2	Regression Execution - Hours spent /Cycle	NA	NA	NA	24		24	
	% Coverage - Smoke Test Suite	NA	NA	NA	100%	100%	100%	14
	% Coverage - Regression Suite	NA	NA	NA	30-40*%	80%	30-40*%	326
	Cross Browser Testing	0%	0%	0%	0%	80%	0%	20*
App3	Regression Execution - Hours spent /Cycle	NA	NA	70	80	70	80	
	% Coverage - Production Checkout	NA	NA	NA	NA	NA	NA	NA
	% Coverage - Regression Suite	NA	64%	69%	70%	85%	70%	110
	Cross Browser Testing	0%	0%	0%	0%	80%	0%	20*
App4	Regression Execution - Hours spent /Cycle	240	240	150	100	70	100	
	% Coverage - Production Checkout	70%	100%	100%	100%	100%	100%	36
	% Coverage - Regression Suite	90%	90%	90%	90%	90%	90%	154
	Mobile Automation	NA	NA	NA	NA	70%	50%	35
	Cross Browser Testing	0%	0%	0%	0%	80%	0%	20*
App5	% Coverage - Regression Suite	NA	NA	NA	NA	90%	20%	
	Cross Browser Testing	NA	NA	NA	NA	80%	0%	
App6 - Mobil	% Coverage - Regression Suite	NA	NA	NA	NA	90%	0%	
App7	Regression Execution - Hours spent /Cycle	NA	NA	NA	20	30	20	
	% Coverage - Production Checkout	NA	NA	NA	NA	NA	NA	
	% Coverage - Regression Suite	NA	NA	NA	40%	60%	40%	8
	Cross Browser Testing	0%	0%	0%	0%	80%	0%	20*
App8	Regression Execution - Hours spent /Cycle	NA	NA	NA	NA	20	NA	
	% Coverage - Production Checkout	NA	NA	NA	NA	NA	NA	NA
	% Coverage - Regression Suite	NA	NA	NA	70%	90%	70%	30
	Cross Browser Testing	0%	0%	0%	0%	80%	0%	20*
App9	Regression Execution - Hours spent /Cycle	NA	NA	70	100	50	50	
	% Coverage - Production Checkout	NA	NA	NA	100%	100%	100%	22
	% Coverage - Regression Suite	50%	50%	50%	88%	88%	88%	93
HCCE	Regression Execution - Hours spent /Cycle	NA	NA	NA	NA	NA		
	% Coverage - Smoke Test	NA	NA	NA	NA	100%	70%	7
	% Coverage - Regression Suite	NA	NA	NA	NA	70%	0%	0

Figure 9: Automation Penetration Tracking (year wise)

3.4 Risk Based Testing

Risk Based Testing (RBT) is a technique used to prioritize the development and execution of tests upon the impact and likelihood of failure of the functionality or aspect being tested.

RBT focuses test activities on tests that will mitigate the greater risk. It creates an optimized and prioritized test suite with fewer test cases, and the ability to uncovers most of the defects. It also Increases test effectiveness while potentially resulting in fewer tests, thus reducing cost while maintaining or increasing test effectiveness. It provides a quantitative report of Software Risk that enables better decision making. We can create a focused approach on Critical User Requirements at each testing stage resulting in easier acceptance of software by customer. Risk based testing approach eliminates ambiguity in status of test project and injects confidence into the process. Figure 10 shows the Risk Based Testing process:

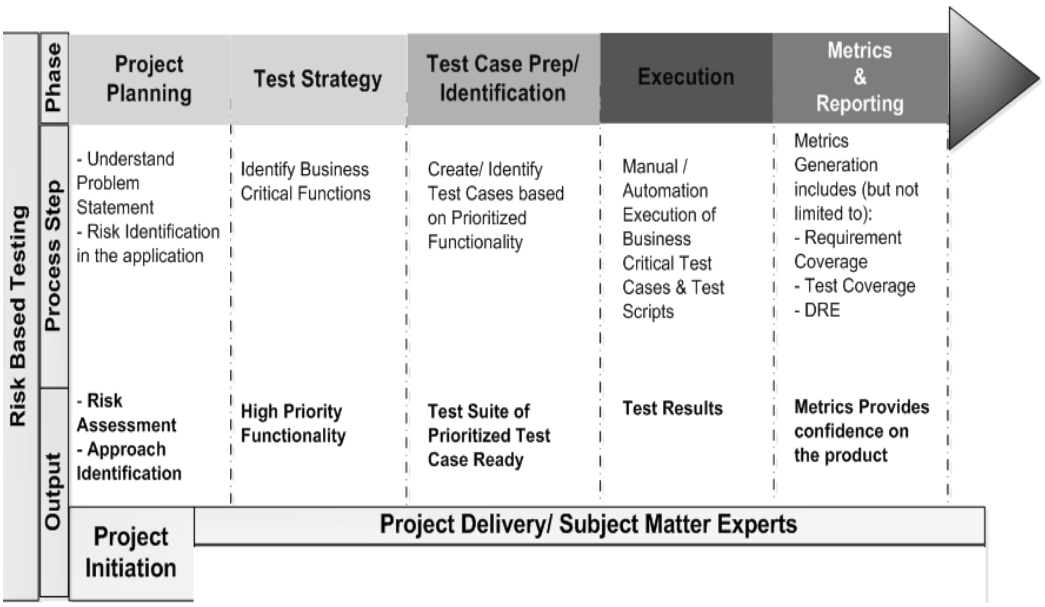


Figure 10: Risk Based Testing Process

Recommendations –
The flowing scenarios are recommended for doing RBT as they give many benefits:

- Applications having critical defects being found late in Test Execution
- Application which has a large test suite.
- Applicable when project timelines at risk.

Case Study:
We were facing a couple of challenges; crtical defects have been found at the later stage of the System Testing Execution cycle and due to that cycle time, were increased beyond the time limits. Another challenge is, when Cycle 1 time is increased, we were not able to execute all the TCs in cycle II and a big risk was involved.

Action Taken:
To mitigate the above challenge, Risk Based Techniques were implemented on one the portals projects; this was a 2000 man Hrs effort for QA to complete that.

The following approach is followed:

1) Prioritized Test Cases as per business requirements priority.
2) Executed test cases starting with High Priority Test Cases and then move towards Medium.
3) Low priorities TC were excluded from execution.

Results:
Below are the benefits from RBT implementation -

- High priority test cases detected 83.3% of the defects.
- Major defects with High Priority Test cases were identified early in life cycle.
- As we move from High priority TC to Medium priority test cases; the severity and count of defects were less.
- There were no UAT defects uncovered for Low priority test cases.

- 256 test cases out of 304 were executed saving 72 hours of execution time which in turn saved $5544.

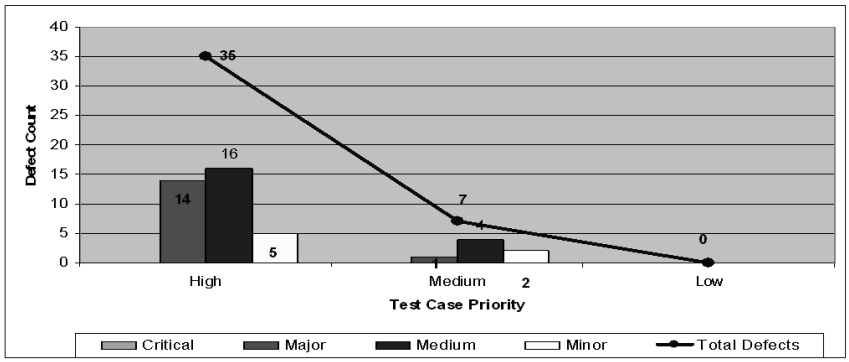


Figure 11: Risk Based Savings in Hrs

4 Productivity Improvement Areas

Productivity Improvement is another key contributor for QA cost reduction. There are many ways to improve productivity:

- By providing more domain knowledge to the team members so that they can do their jobs faster and defect free.
- By Reducing wastage in the system or
- By doing Process Optimization

Techniques

1) Best Practices of Project Management (Resource Optimization)

Cost is directly linked with number of resources/associates put on the project. 85% of the project cost or budget is consumed by the human resources. It is very important that how we are utilizing the human resources, are we enabling them with proper work environment, needs of the employees on day to day basis. In software testing area, tester's have to wait for external dependencies like code deployment, defect fixing, test environment un-stability and so on. Also, it is very important to know how the testers are deployed on the project.

Below are the very useful techniques which can help the tester or dept to be more productive:

- Minimize Wait Time
- Resource Optimization (On-loading/Offloading)

2) Common Resource Pool

3) Productivity Metrics

4) Enhancing Domain Knowledge

4.1 Best Practices of Project Management (Resource Optimization)

4.1.1 Minimizing Wait Time

As a QA organization, we often deal with down time. QA have to wait on day to day basis for different external reasons such as waiting for new code for testing, defect fixing, issue resolution, requirements clarification, environment availability, database down, test data availability and feed from upstream applications, and so on. This list is endless and delays from any of the above puts lot of pressure to complete Test execution in allotted time. In real world, it is not possible to eliminate Wait Time totally, but by using few best practices this can be reduced substantially, thus helping in overall productivity.

The main reason for QA downtimes are:

Environment Unavailability: This is due to daily build deployments in regular hours when the whole team is working and it goes on for approx. 30 mins or more, sometime deployment fail, Code Merge issue, and so on. Also, scheduled downtimes of the back-end systems resulted in approx. 6-7% of waiting time for these activities.

Incorrect or Unavailable Test data: Especially in the case of upstream applications, they have to depend on the test data on the downstream applications. They send their test data request on time but data is not delivered on time and, most important, test data is not setup properly or correctly. That resulted into more re-work to correct the data and increase the waiting time for the upstream applications. Incorrect test data leads to more defects and Dev/QA/SA teams to put more effort to triage the issue.

Delayed Code Delivery: This is not new to IT Industry; in Waterfall or V model, most of the time code delivery is delayed and if it is on time, then code quality is very poor. Testers have to wait at least 3-4 days until code/application has stabilized for smoother testing. It has been observed that in the first week of major project code delivery, most of an application's new major functionalities are not working and the tester ends up doing Smoke testing in the initial few days.

Defect Management: This is another major area where testers have to wait for the resolution of the defects, and block them to do further testing. Most of the time critical defects are resolved in 48-72 hrs. Major defects can block a tester for a week and so on.

The Figure 12 shows the snapshot of defect aging analysis:

Test Phase	Severity	Average Fix Time	Age of Closed Defects	Age of Non-Closed Defects
⊟Assembly Test	1 - Critical	5.67	7.50	
	2 - Major	7.00	9.73	
	3 - Medium	10.00	12.13	
	4 - Minor	1.00	6.60	
Assembly Test Total		**6.55**	**9.44**	
⊟Regression Test	1 - Critical	9.00	10.00	
	4 - Minor	16.00	17.00	
Regression Test Total		**12.50**	**13.50**	
⊟System Test	1 - Critical	2.60	4.20	
	2 - Major	7.94	9.47	33.50
	3 - Medium	4.50	7.53	22.50
	4 - Minor	4.52	5.84	
	5 - Enhancement	0.00	2.00	
System Test Total		**5.20**	**7.51**	**25.25**
⊟Test Planning	1 - Critical	7.00	7.00	
Test Planning Total		**7.00**	**7.00**	
⊟UAT	1 - Critical	0.00	1.00	
	2 - Major	4.44	4.51	4.08
	3 - Medium	3.13	3.14	5.95
	4 - Minor	3.67	5.30	6.27
UAT Total		**3.82**	**3.97**	**5.52**

Figure 12: Defect Aging Analysis

We have captured the Data of 2 applications who has lot of dependency and have captured the data related to Environment Downtime, Test Data, Code Delivery and Wait Time. The below Figure 13 shows the depiction of QA cost Vs Waiting time:

Conclusion: More the Waiting time, more the QA Cost

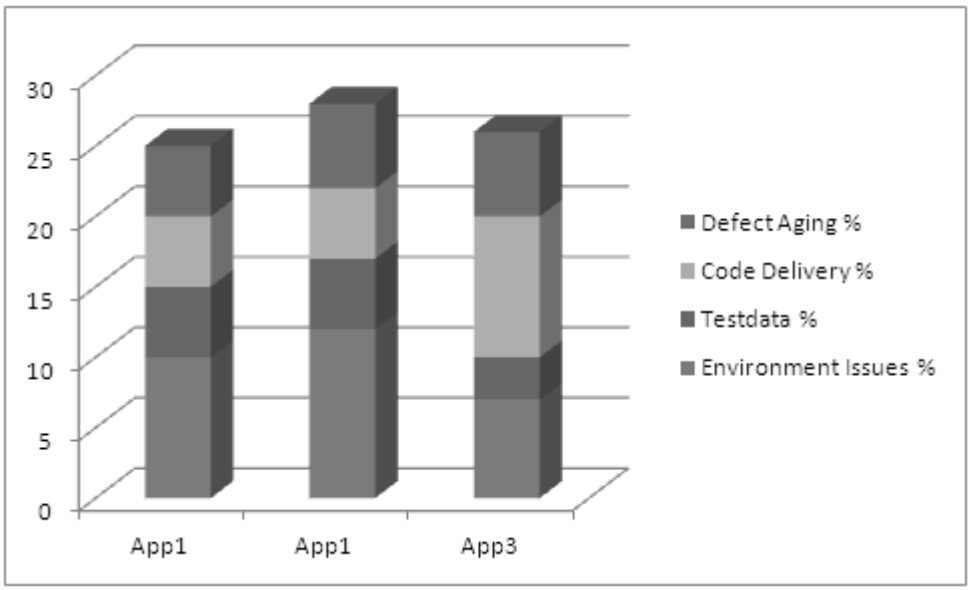


Figure 13: Waiting Time Vs Cost

Case Study:
The above mentioned issues were faced in couple of our application areas and we followed the below good practices to resolve them and hence increase the productivity of the testers.

App Overview: The application is a web based application, with more than 10 Million people in its consumer base. We interact with more than 15 backend systems to get the data from them and display on the portal.

Practice followed
Created an **Environmental Checker Utility**, which helps the testers to know which back-end system is up and running and which is down. Testers can work on other test cases after they have confirmed their environment is up and running. This helps us to reduce non value added defects due to the fact that an environment is down.

Code Deployment is done by the Configuration Management team **during off Hours** (Either early in the morning or late in the evening when testing team is not at work).

Quality Gate process is established for Code handover to QA. **Assembly/Unit testing results** are shared with the QA team before start of the test execution phase. This results in less defects during the system testing phase.

Defect Service Level Agreements (SLA) have been defined to reduce the Defect Aging.

- Showstoppers - ASAP
- For Critical defects – Within 24 Hrs
- Major Defect – 36 Hrs
- Medium Defects – 48 Hrs
- Minor – 72 Hrs

4.1.2 Resource Optimization (On-loading/Offloading Model)

It is a basic principle of project management that **"resources/associates should be aligned to work, not work being aligned to associates"**. The project managers have to identify the skills of their team members and, accordingly, associates should be aligned to work. There are

different phases in the STLC (Software Test Life Cycle) and we know that in each phase we do not have the same amount of work, so it is imperative that the Project Manager perform due diligence in assigning the tasks to the associates. Suppose we get a 100 Man hours project that needs to be completed in 100 days. Does it make sense to put a single associate on the project at day 1? Does it make sense to do this if we know that during the initial and end phases there will be little, if any, work to be done? So, effective resource deployment is the key on project to reduce the QA cost down.

Task allocation is very important. If we start allocating one project/task to an associate, we will see a challenge when the task is blocked. The associate will sit idle and it will add up to Testing Cost. As a best practice, one needs to assign more than one task to one associate so that, if they find themselves blocked due to issues or instability, the associate can work on alternate tasks. By doing this, we can use resources/team members to maximum capacity. With one task, the associate may have work of 35 hours per week against a capacity of 40 hours, but with multiple tasks, we can achieve even 100% capacity utilization.

Another good practice is to have some low priority OPEN tasks already available so that in case of available bandwidth, we can start allocating those to associates. Regression suite creation, automation, and process improvement are good examples of such task.

It is also seen that Project Managers try to block the associates; when things do not happen as planned, often the idle time increases. It is a good practice to load the resource plan so that we have less people during design and development, and get full strength during execution. It's especially important to have extra hands during the last couple of weeks before release. We should free resources and team members if we see delays in requirements, design and code turn over stages.

Resource On-loading/Off-loading Model

(Resource/Associate hiring/engagement as per actual schedule)

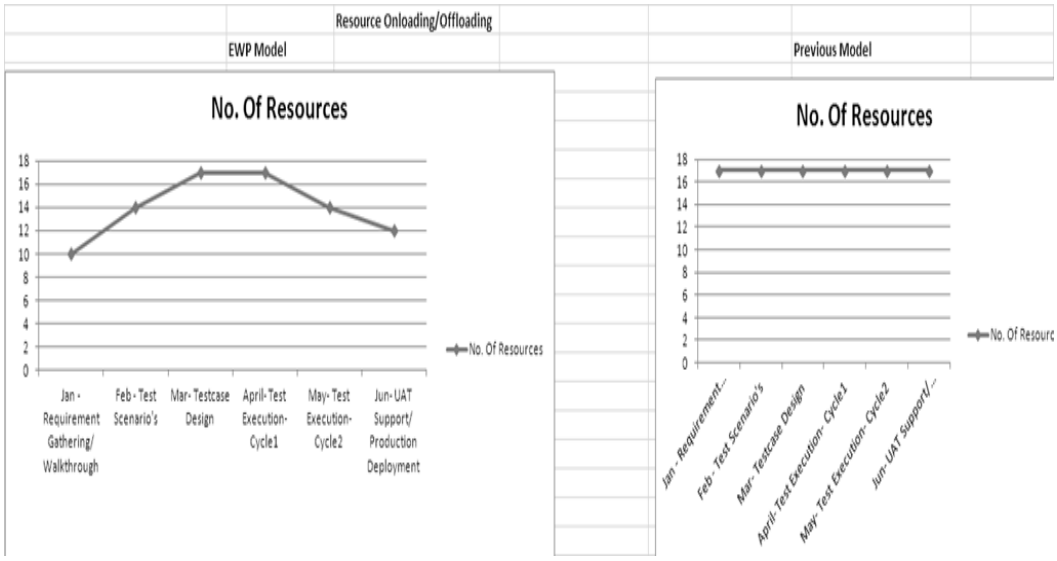


Figure 14: Onload/offload Model

4.2 Common Resource Pool

Generally, in case of big projects, the project team hiring gets started on the basis of the Order of Magnitude (OOM) estimates and associates are onboard when we have less work on the project

but have fully funded project hrs to burn. Managers/leads believe that, according to the resource plan and task alignment, associates are quite occupied in the work. That's not the reality. Associates being charged to the project and capacity utilization is also very good, but at the same time no one looks at the productivity. Even in normal Test Design or Test Execution, even for scheduled down time, attending meetings or other activities, the Hrs are charged to the project that increase the QA cost.

The solution here is to encourage the associates to enter the actual time in the timesheet or Daily Tracking Sheet, which will help the Leads and Managers to know which associate is occupied or not and start building or capturing the Productivity metrics. Also, Leads/Managers can assign some additional tasks, engage them into new initiatives in case associates have some bandwidth.

Case Study

Let us take a look at another real world scenario, where a Code Turnover delay for Testing increases the Project cost exponentially and how this can be minimized. For the sake of simplification, we have taken 2000 hours of Testing effort during our project, in which 10 Team members are required for 5 weeks. In the first scenario, 20% of progress has been made in the first 3 weeks of the project due to changing business requirements and/or ambiguous business requirements. To finish the balance of 1600 hours of Testing effort, the project will need 20 testers which will make overall Testing Cost 40% more than initial estimates.

There are 2 options available to manage this effectively. The first one is, instead of deploying all 10 people in the initial weeks, why not deploy 4 people to project and when the things come in shape during last 2 weeks, and deploy the full battery of 20 people to the remaining 80% of Testing effort? This will bring down the cost marginally more than estimated. The Question is "How will you know that first 3 weeks of project will be able to give you only 20% of effort?" This can be judged based on the maturity of the organization, past experiences and early signs of the project.

The other option is to use Test Optimization techniques like Risk based Testing and OATS – Orthogonal Array Test Strategy Techniques to reduce number of Test Scenarios/Test Cases to reduce scope of Testing so that with limited number of associates one can manage MOST OF THE COVERAGE and NOT FULL COVERAGE.

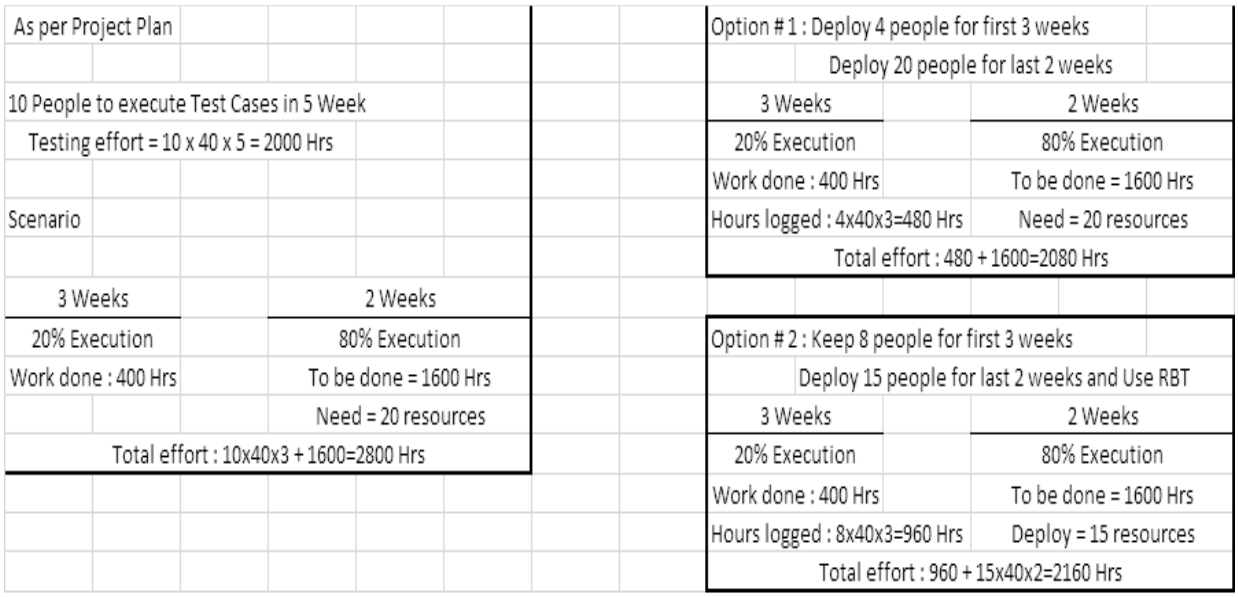


Figure 15: Sample calculation to show Cost increase due to Delays

There is also an important theory that projects expand or compress their actual effort based on given effort. If as an organization you feel that your Testing Team can be more efficient, then the Finance way of doing things is to cut Estimates or effort up front. For a 2000 hours of Testing effort project, Finance can mandate to make a 20% cut, hence allocating 1800 hours of Testing effort. It is now up to the Testing team to manage the delivery ON TIME, ON QUALITY and ON BUDGET. They have to focus on using Test Optimization techniques like OATS, RBT and also work towards removing inefficiencies or reducing their wait stage.

Resource Leveling Issues

It has been observed that, on average, 50% of Testers time goes to waste while waiting for a build to come or a defect to be fixed. This does not mean that 50% of the time the Testers are sitting idle. They try to use this free time to do Test Data preparation, Test Case writing or doing other miscellaneous activities, but at any time this is not the most productive usage of their time. Another fact is that 80% of Test execution happens in 20% of the time.

In a typical 12 weeks Quarterly Release Cycle, various Testing activities are Test Planning, Test Design, Test Data Creation and Test Execution. Various roles being played among testing communities are Lead, Senior Test Engineer and Test Engineer. If we plot those 12 weeks activities and the contribution or distribution of task among various Team members, this comes out to be as shown in following graph:

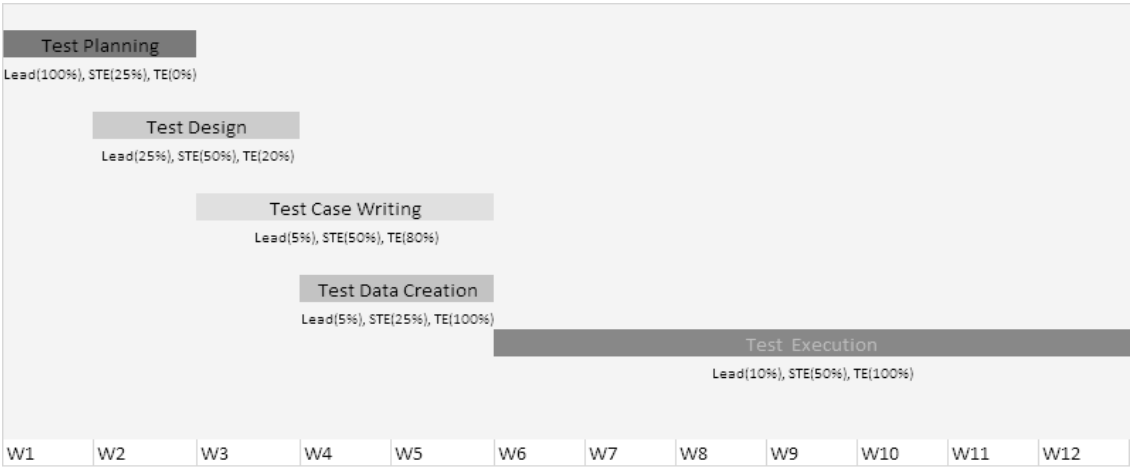


Figure 16: Testing Phase Contributions

If we plot the capacity utilization of all Test team members per role, it comes out as shown in the following graph:

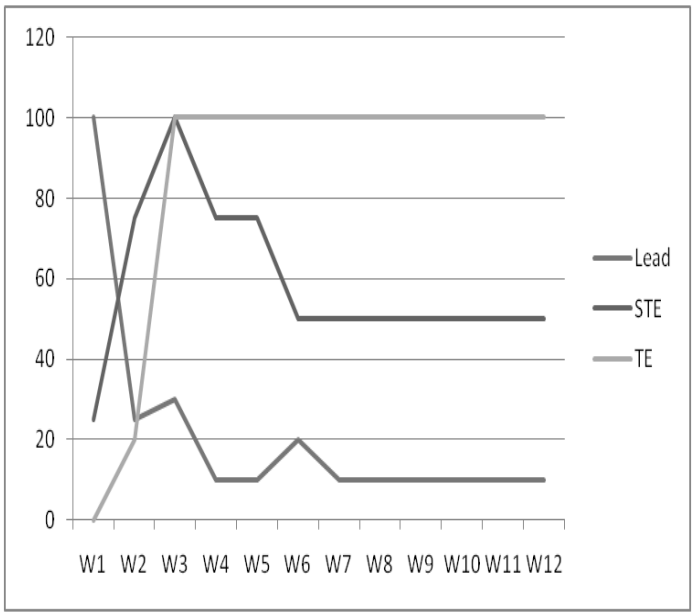


Figure 17: Testing Roles Utilization

The above chart shows

- Leads contribution or usage is more in the beginning, and it slows down considerably after 2 to 4 weeks
- Senior Tester has a varying demand in first 6 months and then tapers down
- Test Engineer need comes week 4 onwards

One should also realize, it is a very common mistake to have dedicated associates (Lead, Senior Test Engineer and Test Engineer) assigned through out the development cycle. This result in

- Cost Overheads
- Inefficiency
- Not deploying the right resources for doing the right task

The simple solution to solve the above problem is

- Tester knowledge sharing across Projects on Time sharing basis
- Develop Specialist for Test Data Management, Test Plan, Test Design and Test Specification
- Junior Testers focus on Test Case writing and execution
- Task Based Planning
- Match Associate skill to Task
- Keep Core Team aligned to project to maintain continuity

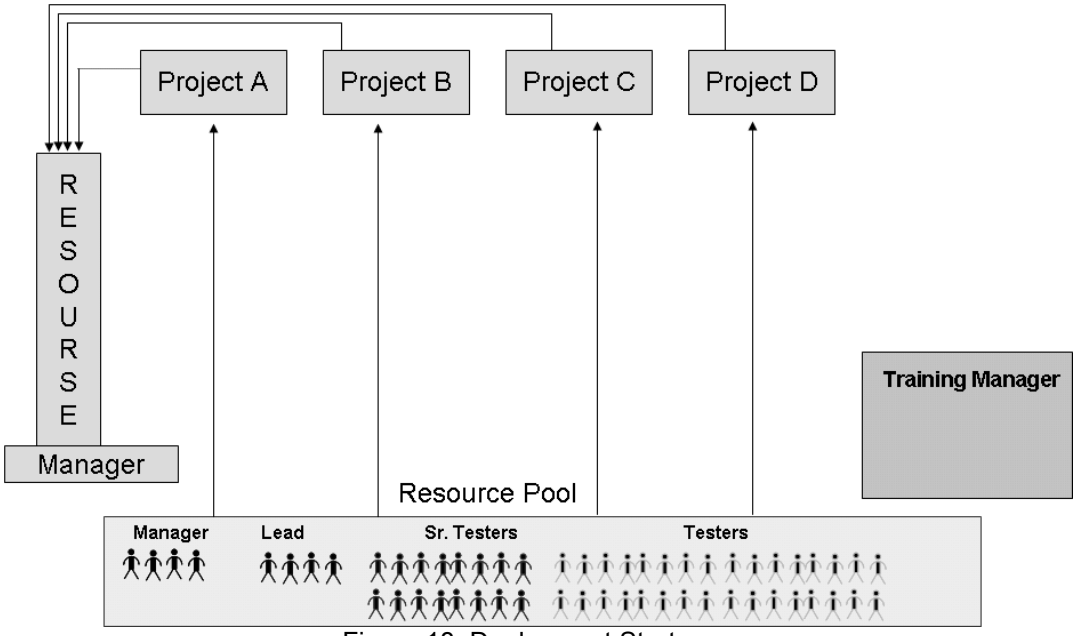


Figure 18: Deployment Strategy

4.3 Productivity Metrics

Productivity metrics are the measurements that help determine what are baseline numbers of different test activities like Test case creation, execution, test data management and so on. From there, the organization or department can set their year on year basis goals. To improve those goals, they can use the above-mentioned techniques like Test Optimization or Productivity Improvement area. Actual Time Reporting is key to get real productivity metrics. It is good if Work Based Structure workflow is implemented in time so that the reporting system and actual time reporting is followed by each individual on the project

It is very important to keep in mind and know what the source of truth is while capturing such metrics. Financial systems or Time Reporting systems should be used for Estimated Hrs and burned Hrs to know the Effort variance and then keep updating the Estimations. Central Test

Management tools should be used to get the numbers of Testcases, Test Execution numbers and Defects numbers.

WBS	% division	Hrs	Remarks
Test Management	16%	142	Management
Create Test Plan	18%	158	Create HLTP & DTP
Create Test Scripts & Data	12%	104	Test data coordination
Execute Integration Test	15%	139	Test execution
Regression Test Scripts & Data Complete	16%	142	Automation
Execute Regression Test	12%	105	Regression testing
QA UAT Test Support	5%	47	UAT support
QA Defect Retest	3%	28	Defect management
Post Deployment	4%	33	Post deployment
Total		900	

Figure 19: Work Break Structure

4.4 Enhance Domain Knowledge

Domain expertise is the key factor in enhancing the quality of a product or project, and hence reducing the QA cost. A tester who has very good domain knowledge can test the application better than others. The domain knowledge plays an important role in software testing as one of the software testing principles says “Testing is context driven”, which means the testing performed for a Financial application is different than testing done for Insurance/heath care application. A tester with domain knowledge will prove to be productive to the project immediately as he or she will be able to understand the problem in depth and come up with solutions. This will in turn add value to the project and the product by ensuring that it meets the quality standards.

Here are a few advantages of Domain Knowledge:

1. Reduces the Training time: A tester can be productive quicker than a person that has little or no domain/industry knowledge.
2. Good idea on UI features: It will help the person for improving the look & feel as we’ll catch more bugs at initial stage.
3. Good idea on Back end Processing: how effectively/efficiently is the data/code being handled
4. Domain knowledge is also important to defect triage: Knowing how the application will likely be used, and how it is expected to perform, will tell QA whether a given defect is trivial, significant, or critical.

To enhance domain knowledge, one needs to establish the baseline or create the current skill matrix and share it with the associates. Once associates know their current level of domain knowledge on a particular application, the improvement goals can be set to track it on a regular basis. The practice shown below can be used to track to the skill set at department level and at an associates’ level on a regular basis.

Module	SME	Advanced	Intermediate	Beginner
Member	0	15	13	4
Provider	7	11	12	2
Claims	9	15	6	2
Billing	5	3	7	17
Commissions	0	2	3	27
Utilization Management	3	1	7	21
Case Management	0	3	3	26
Customer Service	3	4	3	22
Capitation	0	1	1	30
Accounting	0	2	2	28

Figure 20: Skill Matrix

Using Metrics to Drive Customer Satisfaction

Jeff Fiebrich
Freescale Semiconductor, Inc.
j.fiebrich@freescale.com

Abstract

When beginning a metrics program, the most important thing to ask is 'Who are the metric's customers?' Each metric should have a customer that is prepared to take immediate, documented action when indicated by the metric. A metric without a customer is a waste of time.

When implementing software metrics, one must understand the purpose behind the metrics. Is the metric to measure software development productivity, to illustrate the level of quality in software products, or simply to drive an initiative or goal? Development teams will typically want to deliver software productivity metrics to management while management will typically want to see metrics that support the company's effort to increase customer satisfaction - two very different things. To resolve this issue, a discussion must occur that captures what data the customer needs to see in order to react in a way that promotes the purpose of the metric. Once the metric reporting parameters have been established, the logistics of successfully driving the metrics can easily be defined.

Biography

Jeff Fiebrich *is a Software Quality Manager for Freescale Semiconductor Inc. He is a member of the American Society for Quality (ASQ) and has received ASQ certification in Quality Auditing and Software Quality Engineering, and is a RABQSA International Certified Lead Auditor. A graduate of Texas State University with a degree in Computer Science and Mathematics, he served on the University of Texas, Software Quality Institute subcommittee in 2003 - 2004. He has addressed national and international audiences on topics from software development to process modeling.*

Jeff has over twenty-five years of Quality experience as an engineer, project manager, and software process improvement leader. He has lead efforts in ISO certification and Software Engineering Institute (SEI) Maturity and SPICE assessments. Jeff has led improvement efforts in the United States, Middle East, Europe, India, and Asia.

Jeff is the co-author of the book, 'Collaborative Process Improvement', Wiley-IEEE Press, 2007.

Introduction

Metrics programs are all about taking measures and turning them into information. This information is then used to make some sort of decision – whether it is to take some action or to make a process improvement. These actions or decisions then – hopefully – create a better end-customer experience. Keeping the customer in mind, and remembering that you have customers at many levels, when beginning a metric program the most important thing to ask is 'Who are the *metric's* customers?' Each metric should have a customer who is prepared to take immediate, documented action when indicated by the metric. A metric without a customer represents time wasted. And of course, a "customer" can be internal or external; it can be your manager or CEO; it can even be a process.

It is particularly important in implementing software metrics, that you understand the purpose behind them. Is the metric intended to measure software development productivity, to illustrate the level of quality in software products, or simply to drive an initiative or goal? Metrics can have a cause and effect relationship with each other. For example, development teams will typically want to deliver software productivity metrics to management while management will typically want to see the effect of productivity which, in turn, supports the company's effort to increase customer satisfaction. These are two very different things. To resolve this issue, a discussion must occur that captures what data the metrics customer needs to see in order to react in a way that promotes the purpose of the metric. Once the reporting parameters have been established, and the interrelationships understood, the logistics of successfully deriving the metrics can easily be defined.

Establishing Metrics

Establishing metrics in today's environment is challenging and time sensitive. It used to be that a typical improvement effort was discussed in January, implemented in March, and checked in November to determine success. Today, metrics customers are requesting improvement efforts in an eight to sixteen week timeframe. They are requesting weekly status reports.

Even if you have identified the metrics you want to analyze, you must ensure that they are based on current data. Data that is a month, week, or sometimes even a few days old can initiate a metrics customer to take an action that is no longer warranted. Real-time data is preferable, and a desirable feature is to have on demand capabilities from a utility like SharePoint's Performance Point, or other business intelligence dash boarding tools. Of course, there are many steps that must be taken prior to establishing an on-demand metric.

We can also take the typical annual improvement program to resolve an in-house issue and compact it down to a few weeks. Yes, this acceleration of effort will require an increase in resources, but a well-executed metric can be used to show management that an issue that was scheduled to be resolved in December was resolved in March; and another was resolved in April. Early resolution of an issue should always be seen as a financial win for a company but remember, your metrics customer must be engaged and have a stake in establishing that the correct data is tracked.

A properly established metric can assure you and your customer that progress is being made and that they are on schedule. If there is a possible issue, everyone will be aware of it at the earliest possible moment. In this way, you can capture and correct issues long before they arrive in your end product. This, in turn, will ensure a better resulting product!

1.1 A Metric must have a customer

Each metric must have a customer! Even the latest and greatest metric being publicized by the top software blog may be meaningless when it comes to your own customers' needs. Often, the data that we gather, analyze, and report in the coolest graphic is without a customer. Without a customer that 'wants the metric', 'has requested the metric', or 'must have the metric', the most amazing metric is irrelevant. Metrics that are in-demand by their customers are very powerful.

1.2 Metrics can support your company's goals

Metrics can easily track your company's success at achieving its' goals. Most industries sit down in January and establish a set of goals for the year. Many times, employees' bonuses are tied to successfully meeting these goals. The last thing you want to do is have updates on a quarterly or semi-annual basis. If one can establish a metric that can illustrate the current status of achieving bonus goals, metrics customers will be waiting in line to utilize it. [1]

Often your metrics customer will not see the connection between software productivity, and end customer satisfaction scores. It will be your particular challenge to show them how metrics interrelate and contribute to end product quality. Once they can make the connection, they will help you establish a more targeted metrics program that may include customer satisfaction goals as an aggregate of other metrics goals.

When annual goals are established, a company may publish five to ten goals and it is tempting to try to supply metrics for all goals at the same time. Take it slow. This would be overwhelming for you and your customer, like trying to boil the ocean. Instead, pick a goal that is being stressed by a potential customer as an important goal. Develop and present a high quality proposal and example for the potential customer. During your discussion, explain how the metric aligns with the customer's goal. If you do this effectively, your customer will come back for more.

Don't fall into the trap of pleasing your customer to get them on your side. Giving a customer a metric that tells things that you want him to know is usually wasted time, even if he says he likes the metric. Focus on demonstrating how your data can be used to show how he is progressing with his priorities. The amount of effort required by you to provide the metric is irrelevant. If the customer experiences the positive effects of a metric, he will come back for more support, and maybe next the metric will seem more valuable to everyone. This usually requires many discussions and demonstrations before a partnership is fortified, but it will happen!

1.3 The outcome of metrics

The metric supplier and the metric customer must understand the actions that the metric customer will take after reviewing his metric. This information is typically documented in the metric's Meta Definition. It is important that the customer understand that you are giving him metrics to enable him to drive improvement/success; not to review during his morning coffee. For example, upon receiving a metric report that indicates progress, the customer may be able to loosen or relax previous commitments or even restructure resources. Likewise, given a metric report that indicates slippage, the customer may immediately devote more of his time to this activity or even request immediate corrective action put in place. The possible reactions of a customer will greatly vary, but each reaction should be documented in the Meta Definition. A metric report that does not generate an action by the customer should always be questioned. As the metric supplier, you should monitor the customers' reactions.

Implementing Metrics

Good things happen as a result of good metrics. If you generate a metric week after week that results in no action taken, then you are not generating a metric that brings value to the table. One good test of a good metric is to quit generating it. If no one misses the metric, it was of no value.

As metric suppliers, we always fall in love with our metrics like parents with their children. And many times we will be the last person to acknowledge that our child (metric) has become completely worthless. Even when 'we' evaluate 'our' metrics annually, we fail to recognize that the metric is no longer performing for its customer.

1.4 Create the Meta Definition

Creating a metric is like many other tasks in life; planning is essential! With that said, it is also important to understand that every metric can be improved and no metric is perfect. Simply take your best shot at the metric to the potential customer and then fine tune it toward perfection. They may quickly find out that perfection is the enemy of good, and learn to accept change as a normal part of the metric program.

During the definition phase, the team must create a Meta Definition for each metric. This is where the owners of the metric will document how this metric will be created and used; refer to Table 1. This will most likely be the most painful stage of defining a metric. As implied in the Latin definition of 'Meta', *beyond*; the level of detail captured, documented, briefed to the team, and approved by the team should be beyond any level previously imagined. Although it is possible to spend too much time in the definition (beyond) phase, this is rarely the case. An example of a typical Meta Definition can be seen below. This will be the Bible for the metric. [2]

Table 1 - Meta Definition

Download file:	*Link to the file for download*
Operational Definition	
Metric Name	*Metric Name*
Metric ID	*Metric Unique ID*
Goal	*Goal derived from management or other information needs, that the current metric contributes.*
Purpose	*Purpose of the metric being defined, in relationship to the above mentioned goal.*
Computation	*Formula*
Unit of measure	*KLOC, %, etc.*
Data items	*Detail and explain all members of the formula.*
Data source	*Where to collect from the values of the formulamembers.*
Tools	*Tools, templates, checklists, scripts etc. used to compute metric.*
Phase and frequency to collect	*Frequency of collection - weekly, monthly, quarterly, yearly, on a need basis - specify what phase of the project when the metric must be collected.*
Responsible for measuring	*Who is responsible for measuring?*
Interpretation/Actions	*How to interpret values? What infers a positive status; a negative status? What are the possible actions to be taken each time a positive or negative status is detected? (also see target and threshold section)*
Target	*How to set target; value of the target if known.*
Threshold	*How to set threshold; value of the threshold if known.*
Reporting to	*Where to report the measurements?*
Frequency to report	*Frequency to report.*
Responsible for reporting	*Who is responsible for reporting?*

Recommended action on deviation	*Action recommended when deviation from target occurs.*
Methodology to collect	*Who is doing what; from where, store where; etc.?*
Measurements storage place	*Where to store measurements?*
Integrity verification	*How measurements integrity is verified?*
Responsible for integrity verification	*Who is responsible for integrity checks?*
Reference:	*References, standards etc.*
Analysis of measurements data	
Analysis spec and procedure	*Specify and prioritize the analyses that will be conducted and reports that will be prepared.*
Data analysis methods and tools	*Visual display, presentation techniques, descriptive statistics (mean, median, mode).* *Statistical sampling criteria, analysis in presence of missing data elements; analysis tools.*
Administrative	*Administrative procedures for analyzing data and communicating results. Approach to review and update analyses and reports. Approach to update measures as necessary. Criteria for evaluating: utility of analysis results and for evaluating the conduct of the measurement and analysis activities.* *For example, results are provided on a timely basis, understandable, and used for decision making.*

1.5 Get full participation

One of the important logistical actions will be assuring that all data points are included. If we are reporting to the CEO the on-time delivery to external customers, this metric must include the status of every delivery to every customer. We cannot selectively omit the status of deliveries to customers who are in payment arrears. The reality is that these might be the most important deliveries for this metric.

A good rule of thumb is that a metric reported at the group level must have everyone in the group participating. A metric reported at the division level must have everyone in the participating division.

1.6 Agree on the metric report format

The supplier and the customer of the metric must agree on the metric report format. It is best to include an example of the report in the Meta Definition. The driver of the format should be the customer. After we go away and gather the data, clean the data, format the data exactly how we believe the customer wants it, and apply some graphic sparkle to the data just like we heard the customer request, it can be very difficult to hear the customer say 'this is not what I wanted.' Patience will be the winning virtue! Continue to have short discussions with the metric customer until the customer is completely satisfied. Many times, if the customer does not feel he has designed the metric, he will not embrace the metric. You want him to feel like he is the owner of the metric.

1.7 Check your emotions at the door

As we all know, given a bucket of data, we can easily take that data and present it in a manner that supports terminating a project and extending the project at the same time. Even with a great Meta Definition for your metric, different individuals will have a different emotional investment to the data.[3] Using multiple individuals to collect and crunch portions of the data rarely works. The easiest way to avoid metric emotions is to have one person crunch and beautify the data. Establishing a script or query

to accomplish the majority of this work is even better. The less human intervention the better. It is likely that none of the individuals represented by the metric data will appreciate this effort, but it is only the customer's appreciation that is important to a successful metric.

Using Metrics to Drive Customer Satisfaction

Metrics must always be used for good, and never evil. This is easily said and yet never believed. Let's focus on using metrics to drive customer satisfaction!

1.8 Defect Count

All of our end customers want zero defects in their products. How can we convince our customers that we are seriously chasing this unrealistic demand? Any company that is satisfying the incredible schedule of its software deliveries and is profitable will likely never achieve the zero defects demanded by their customers. Many customers are happy to accept a percentage goal reduction over the previous year's defect count. For example, have a 20% reduction of defects over the previous year. In other words, if a customer experienced ten defects during the previous year, they are surely not pleased with your performance. If you tell him that this year you have set a goal to deliver him eight defects in his product, he should find you extremely frustrating. Why would he be happy to know that you plan to impact his production eight times this year? However, if you tell him that you have set a goal of a 20% defect reduction over last year, it is much easier for him to envision this number going ***to zero*** at some point in the future.

Figure 1 shows an example of the effect of a 20% reduction of Severity 1 and 2 customer defects. This metric report does not show when *Zero Defects* will be achieved. What it does illustrate is a *Drive Toward Zero Defect*.

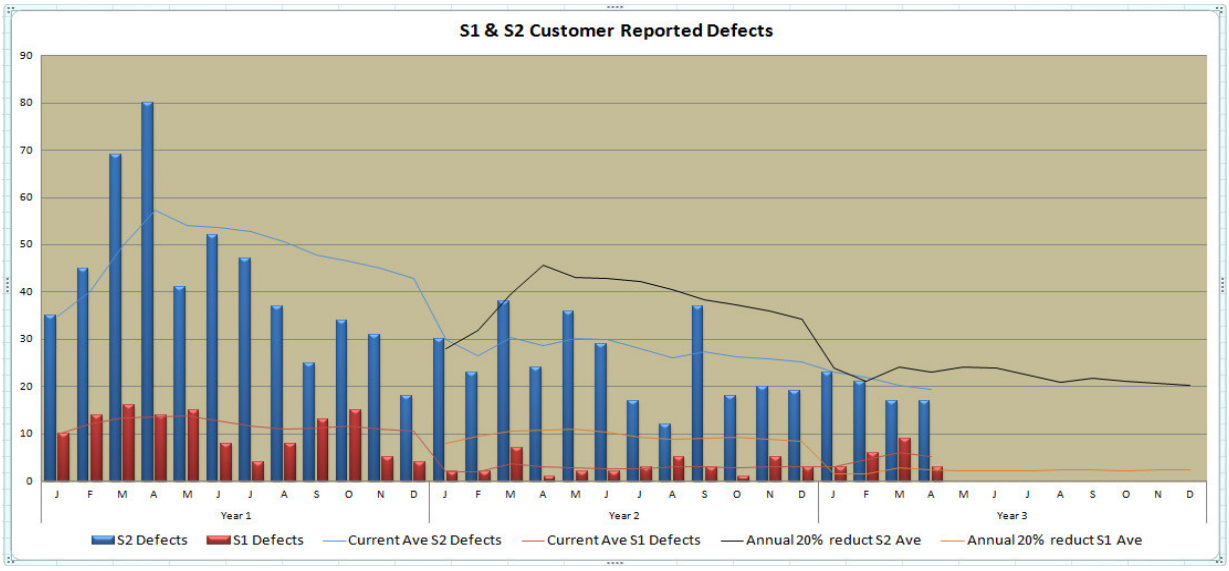


Figure 1. Reduction Report

Often a specified percentage reduction of defects is more easily embraced than an arbitrary number. The percentage reduction can easily be tracked weekly, monthly, quarterly or annually and reported to your customer.

1.9 Defect resolution cycle time

How quickly a customer's defect was resolved is an excellent metric for driving customer satisfaction. Like all metrics, the Meta Definition for this metric leaves space for interpretations. Can it be said that the defect is resolved when the customer receives the updated component or when he is told how to patch his component or why the defect occurred?

When addressing defect resolution cycle time, consider what will make the customer happy. One customer may require a forty-eight hours cycle time, while another customer may just want the update in the next software release in six months. So if it takes you seven days to resolve a defect, you will have to know the needs of the customer to determine if the seven day resolution will make him 'satisfied.'

1.10 Defect Root Causes

Another example of a metric used to drive customer satisfaction is a Defect Root Cause metric.

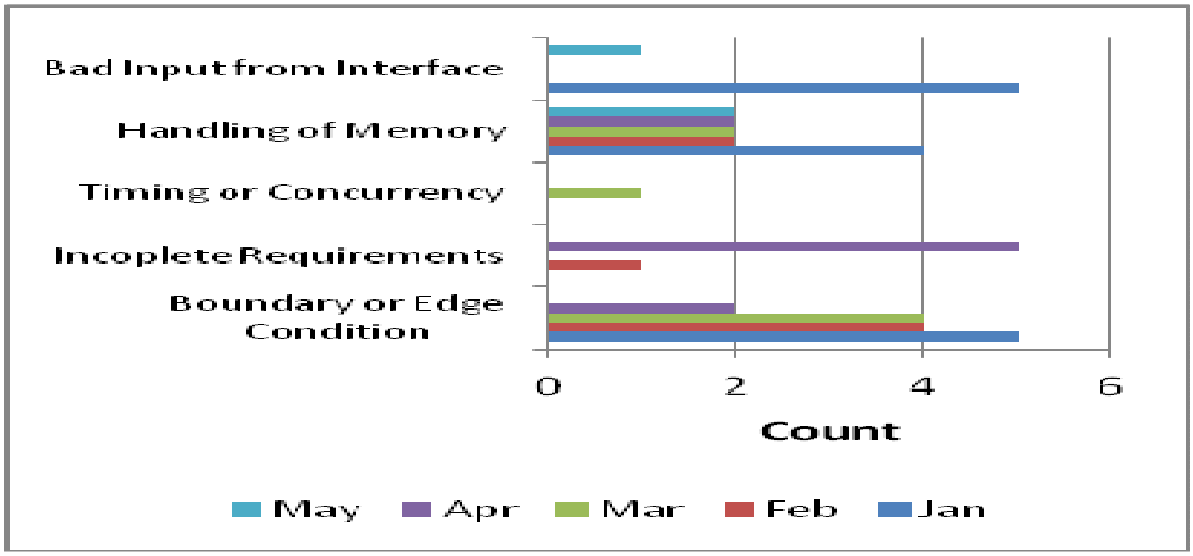


Figure 2 – Root Cause Histogram

Simply assimilate your customers' defects and determine the root cause of each defect. Place the root causes in a histogram as shown in Figure 2. On a regular basis, review this metric, document, and implement a plan of action to prevent the top three root causes from recurring. The action should be designed for immediate impact. The histogram shown in Figure 3 illustrates effectiveness of plans of action put in place for *boundary or edge condition* root causes.

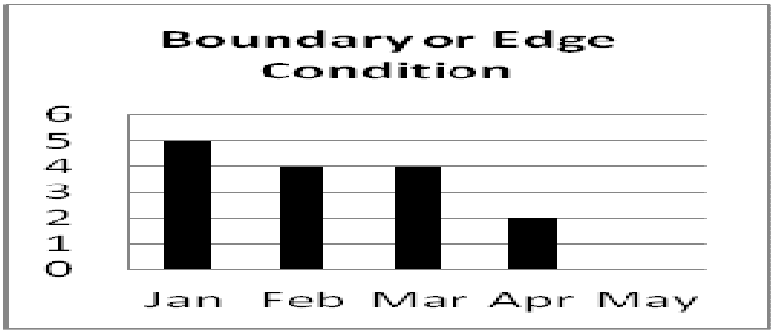


Figure 3 - Boundary or Edge Condition Histogram

On the other hand, if you execute a plan action for four months to resolve the *handling of memory* root cause and this defect continues to be delivered to your customer as illustrated in Figure 4, a serious problem exists. It is obvious that a more effective plan of action should have been put in place? Either the requirements in the Meta Definition were not followed or the Meta Definition Interpretation/Action

section needs to be updated to prevent this from occurring. The Meta Definition will also need to be reviewed, approved and briefed to the team each time it is updated. Any time the metrics indicate that an Interpretation/Action is necessary but this indication is disregarded then escalation may be necessary.

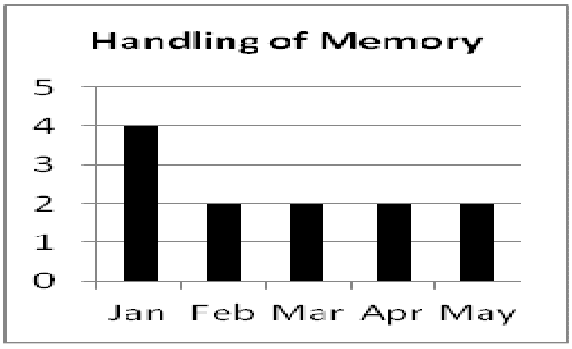


Figure 4 – Handling of Memory Histogram

1.11 Using Multiple Metrics

Often a combination of metrics can be used to show a more convincing story. For example; are we increasing customer satisfaction? The Figure 5 shows that our customer's defect count is below our goal and that should make our customer happy. *(Customer: "Can that be improved?")* The defects are largely being resolved within the fourteen day resolution cycle requested by the customer. *(Customer: "Can that be improved?") But* what are we doing to continually improve these numbers? Well, we have assimilated the root causes of all the defects; picked out the worst offender; and put in place effective action plans that permanently resolve them. *(Customer: "Oh. Thanks!")*

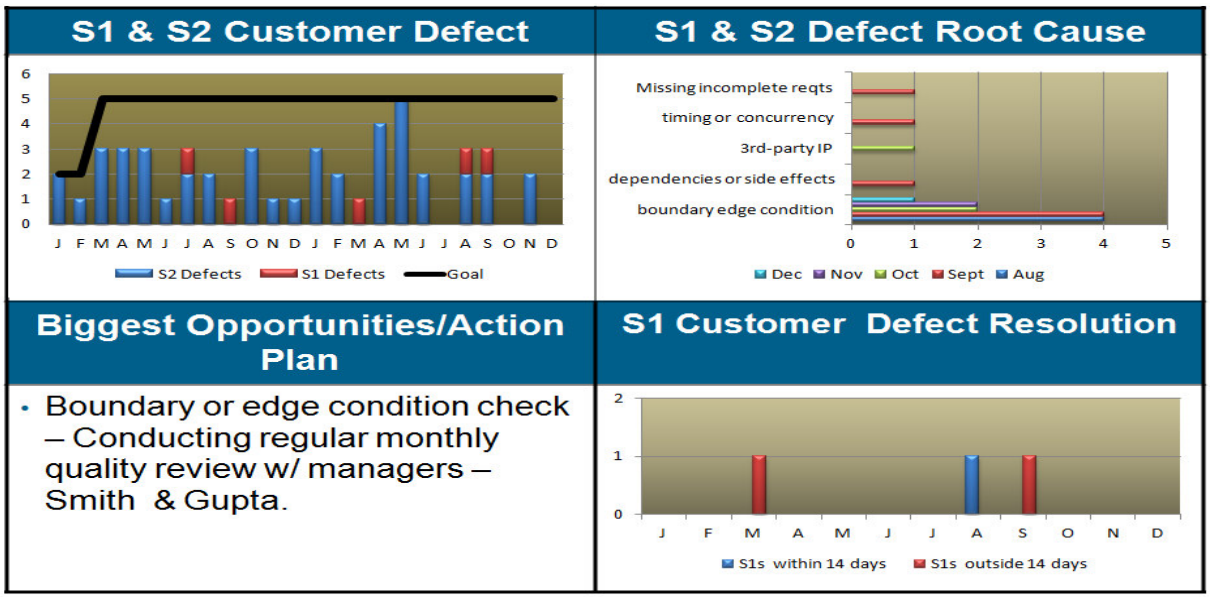


Figure 5 – Combonation of Metrics Example

Figure 5 is an example of a graphic that can be used to assure a customer that your daily goal is to focus on satisfaction. It also shows where resources can be directed to create the greatest impact.
Although metrics are a great way to get a feeling of your current status, if a metric is several years old, it can be good to illustrate that the metric is not stagnant, but is constantly being used to drive-up product quality. In Figure 6, the three metrics have been in place for over two years; but the goals continue to tighten. For example, last year (Last Yr's Exit) the percentage of S1 and S2 Customer Reported Defects (CRDs) was reduced 15% over the previous year; this year our target (This Yr's Target) reduction goal is 20%; currently our year to date (YTD) reduction is 32%.

Group 1		Last Yr's Exit	This Yr's Target	Current YTD	Comment
Category	KPI				Status Notes
Customer Defects	% of S1 & S2 CRDs vs. Total Defects	12%	10%	4%	(S1 & S2 Customer Reported Defects/ total number of valid defects)
Customer Defects	Reduced % of S1 & S2 CRDs over 2011	15%	20%	32%	S1 & S2 Customer Reported Defects
Customer Responsiveness	% of S1 CRDs resolved within 14 days	86%	100%	92%	S1 Customer Response < 14d

Figure 6 – Long term metric example

Summary

A good metrics program will consider the timeliness of the data used, which should be defined by its intended usage. This means that the metric customer should be engaged from the beginning to evaluate what actions they will take if the metric triggers the need for an improvement. Remember, too, that customers come in all shapes and sizes; they may be senior leadership establishing and tracking company goals, the customer establishing delivery goals, or anything in between.

A good metrics program will focus on the following four beliefs:

- Metrics are about taking measures and turning them into actionable results.
- Metrics should be developed in a partnership with the customer and the provider.
- Metrics may or may not align with company goals, but it is a bonus when they do.
- Meta Definition is a useful tool for documenting how metrics will be created and used.

Satisfaction is tied to using the metrics to show the customer that your team and company are working to continually improve the process and the product.

References

1. Yeakley, C., Fiebrich, J., “Collaborative Process Improvement”, IEEE-Wiley Press, March 2007.
2. Fiebrich, J., Garay, D., “What Happens in Guadalajara Will Not Stay in Guadalajara!”, in *Proceedings*, Better Software Conference & Expo, June 2007, Las Vegas, Nevada.
3. Secuiu, I., Fiebrich, J., “Successful Process Improvement in Romania", in *Proceedings*, Central and Eastern European Annual Conference for Software Process Improvement, November 2008, Budapest, Hungary.

Acknowledgements

I would like to acknowledge G. Ashleigh Fails and Moss Drake for their editing assistance.

Metric-driven Fuzzing Strategy for Network Protocols

Faraz Qadri, Vivek Jain, Aniket Malatpure
farazq@microsoft.com, vivekjai@microsoft.com, aniketm@microsoft.com

Abstract

Fuzzing is a commonly deployed testing strategy to uncover security vulnerabilities. Measuring coverage and assessing fuzzing effectiveness add significant value to the fuzzing process. Metrics like interface coverage, input space coverage, code coverage (code volume, cyclomatic complexity, defect density, known vulnerability density, tool soundness) have been proposed for the above. However, there is a dearth of industry case studies that show application of such metric-driven fuzzing approaches to standard network protocols. Hence, security practitioners lack step by step guidance to define metrics for planning and executing a fuzzing project.

This paper introduces a metric-driven fuzzing strategy and shows its application to standard network protocols. The strategy is a stage-based approach, which starts by evaluating the system as a black box, to analyze attack surface and attack vectors in common usage scenarios. In the next stage, the system is explored from a white box perspective and blocks of code that consume untrusted data are identified. In the third stage, tools are developed to cover both black box and white box scenarios. Metrics relevant to each stage are created. In the final stage, these metrics become the driving factor to track progress, measure coverage and continually improve the fuzzing process.

We present a case study that applied this fuzzing strategy to three different network protocols (SMB, NFS & iSCSI). Being able to create metrics early in each stage helped our fuzzing efforts effectively meet the exit criteria. This approach also helped uncover different classes of security vulnerabilities that were not exposed by prior methods. Furthermore, the generic nature of this fuzzing strategy enables it to be used as a template for fuzzing most network protocols.

Biography

Vivek Jain is a Software Development Engineer in Test at Microsoft in Windows Sustained Engineering (WinSE) group. He has been working at Microsoft for the last 6 years. His primary focus is releasing security and non-security Windows updates in various network protocols like SMB, Webdav and NFS.

Faraz Qadri is a Software Development Engineer in Test at Microsoft in WinSE group. He has been working at Microsoft for the last 5 years. His primary focus is releasing security and non-security Windows updates in various network protocols like NFS, SMB and iSCSI.

Aniket Malatpure is a Senior Lead Software Development Engineer in Test, in the WinSE group. He has been working at Microsoft for the last 8 years. His team owns shipping security and non-security updates for Windows File Server. He has a Masters in Electrical Engineering from Clemson University.

1. Introduction

Security testing has become an integral part of the testing process especially with the advent of storage network solutions, web services and cloud computing. High availability and data protection requirements imposed on these services make a compelling case for secure deployments across enterprises. This distributed model implies an increased attack surface as critical data now flows over secure and insecure networks. Security vulnerabilities in such networked systems can cause adverse consequences ranging from denial of service to data compromise.

The well-known method of testing a network protocol for security vulnerabilities is fuzzing. In fuzzing, malformed data is fed to the system to detect if the code handling this data behaves in an insecure manner. While the adoption of fuzzing is gradually increasing in security testing landscape, a common question that arises for testers is, are we done with fuzzing? Are we confident enough in our fuzzing effectiveness? How can we measure the value of fuzzing?

In order to reach an accurate conclusion, we must be able to measure fuzzing. Measuring fuzzing, by using metrics, is an important step when it comes to robust fuzz testing. It is also equally important to be able to use these metrics as a key factor in driving fuzzing improvements and hence their effectiveness.

Without this it is hard to reason about fuzzing effectiveness and there is not much improvement we can make in terms of coverage. This coverage comes from a variety of factors such as attack surface coverage, scenario coverage and code coverage.

This paper discusses a metric-driven fuzzing strategy we adopted during fuzz testing. This fuzzing strategy has two primary tasks, performing quantitative analysis, using metrics, to measure effectiveness and continuously improve fuzzing. In order to achieve these, this approach makes metrics the main focus in driving the fuzzing process.

The strategy begins by approaching the system from a black-box perspective. Black-box fuzzing is a method adopted by penetration testers who focus on targeting the various parts of the protocol. These parts of the network protocol constitute the attack surface and need to be covered thoroughly. Therefore, metrics are associated with them to provide an indication on the fraction of attack surface being covered. Examples of such metrics are related protocols, commands and fields, etc. Similarly, a protocol can be attacked in different ways. For example, a network protocol supporting multiple transports has an equal chance of being attacked over both TCP and UDP. These possible attack vectors provide a list of scenarios that needs to be covered.

While black-box fuzzing does provide one with a quick start, it does not however provide any information about the protocol's code coverage details. This is a necessary step in getting an insight into the code paths being executed as a result of malformed packets. Hence, the need for white-box analysis arises. Metrics pertaining to code blocks and selected functions are defined. These metrics not only ensure that critical code blocks are covered thoroughly but also help improve the tool effectiveness.

By using metrics associated with both these approaches in conjunction with continuous tool improvements, one can get a holistic view of what's being covered and how. The sections below will describe each of these steps in detail with examples of network protocols.

2. Background and Motivation

The process of defining metrics for fuzzing is very scattered across the industry. It is crucial to be able to use metrics that measure a concrete aspect of fuzzing and not just an activity. This means assessing the coverage by the number of packets fuzzed or the amount of time spend on execution is not enough.

Many well-defined metrics already exist in the industry. For example code coverage, attack vectors, input space coverage [2], coverage of previous vulnerabilities, process metrics [3] etc. Given the finite amount of time allotted for fuzzing, this list gets quite exhaustive to implement. Ideally, it would be an aid to use just the right mix of metrics, applicable to network protocols, without compromising on its thoroughness.

But because of the dearth of industry case studies demonstrating an end to end approach of using such a strategy, the task of procuring such metrics can be quite challenging. Most of the time is often spent selecting the key metrics, from a huge set at our disposal, and setting reasonable targets.

3. Metric-Driven Fuzzing

Metric-Driven fuzzing is a process where an appropriate set of metrics is defined to track and assess different aspects of fuzzing and then those metrics drive the fuzzing process. This section introduces the metrics sets and describes how development of these metrics and tracking these to completion drives the fuzzing process.

3.1 Attack Surface

This is our first set of metrics. The metrics in this set focus on protocol level details which are published externally. Since the details used to create these metrics cover interfaces, ports, commands for the protocol, we name it ‘Attack Surface’. Attack surface can have these three metrics.

1. Protocol Coverage

2. Command Coverage

3. Field Coverage

For a given component, the Protocol Coverage metric provides an overview of the protocol fuzzing. The Command Coverage metric tracks coverage of individual commands. Similarly, to make sure a specific command is fully covered, within each command, Field Coverage metrics is defined. This way each table contains different levels of information ,ensuring maximum coverage for the level above it. The following tree can illustrate the metrics relationship under Attack Surface.

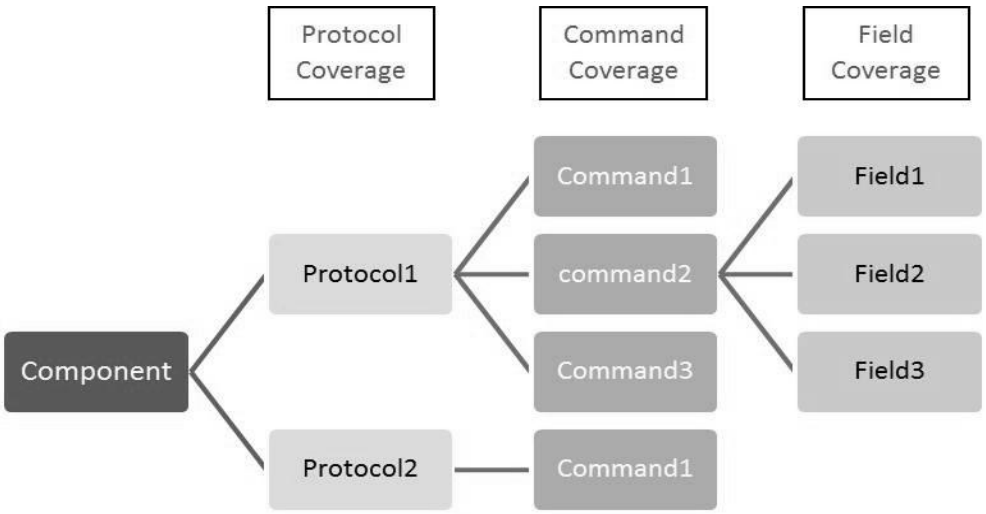


Figure 1: Different metrics under Attack surface

3.1.1 Protocol Coverage

This metric provides a high level view of the fuzzing effort for a given protocol and related set of protocols. Related protocols are those- which use the primary protocol as base (e.g. MS-RAP[4]) uses MS-SMB[5]) but have a unique message format. The list of protocols is prioritized based on the expected severity of a potential vulnerability in the protocol. For each protocol in scope, this table tracks statistics such as number of commands, fuzzed commands and fuzzing approaches. Here is the snapshot of what it should look like.

Protocol Name	Transport Protocol	Priority	Number of Commands (Total)	Number of Commands Fuzzed (Current)	Fuzzing Approaches (Target)	Fuzzing Approaches (Current)
MS-SMB	TCP, NetBios	P0:All protocols processed at kernel level	60	60	MITM* & Generational	MITM & Generational
***MS-RAP**	TCP, NetBios	P1: All protocols processed at user level	19	19	Generational	MITM & Generational
….	…	…	..	..	…	….

Table 1: Protocol Coverage Metrics

***MITM = Man In the Middle**

3.1.2 Command Coverage

This metric set provides a deeper view of fuzzing performed on a protocol by tracking the coverage at the command level. Two metrics are created at this level.

1. Individual commands
2. Command sequences

Individual commands metrics

We categorize those commands based on various parameters like

a) Usage (i.e. Currently Used, Deprecated),

b) Authentication level (i.e. Authenticated, Unauthenticated or Pre-authentication)

c) Ease of generating the command by a user

This helps define fuzzing priority. For example, we would first want to cover currently used and pre-authenticated commands. This metric also captures the Fuzzing tool usage for each command. This data helps the tool coverage metric which we discuss in **Tool Coverage** section. The Minimum Target Iterations count is calculated using the following formulae.

n = number of fields in the command

Multiplier Quotient = Command Fuzzing Priority (1 -10) X 1000

Command Fuzzing Priority is decided based on the role the field plays in overall operation of the command. (10 being most important, 1 being mostly untouched as per protocol spec)

For evaluating the fuzzed values generated by a fuzzer which fuzzes one value in an iteration:

Minimum Target Iterations (N) $= (\sum_{i=1 \text{ to } n} (\#\text{ of unique values generated for field } i))$ *X Multiplier Quotient*

For evaluating the fuzzed values generated by a fuzzer which fuzzes multiple values in an iteration:

n_i *= # of unique values generated for field i*

m = # of fields in command

Minimum Target Iterations (N) $= (n_i \times n_{i+1} \times n_{i+2} \ldots\ldots\ldots\ldots \times n_m)$ *X Multiplier Quotient*

The metric looks like the one shown in Table 2.

Protocol Name	Command Type	Command Name	Fuzzing Approaches	Number of iterations (Minimum Target)	Number of iterations (Current)
MS-SMB	Currently Used	GetDFSReferral	MITM	10000	12000
MS-SMB	Deprecated	SMB_COM_CREATE_NEW	Generational	50000	30000
….	…	…	..	…	….

Table 2: Example Command Coverage Metrics

Command sequences metric

It has been observed that many scenarios cannot be covered when fuzzing is focused on only one command at a time. For those scenarios we define another command level metric which captures command sequences. Target iteration count is calculated based on the same formula as shown in previous section. The metric looks like the one shown in Table 3.

Protocol Name	Command Sequence Name	Number of iterations (Target)	Number of iterations (Current)
MS-SMB	Create-Read-Disconnect	10000	8000
….	…	…	….

Table 3: Example Command Coverage Metrics: command sequences

3.1.3 Field Coverage

This metric captures the lowest level of detail in the black-box fuzzing approach. The purpose of this metric is to make sure a command is fully covered by a fuzzer. The minimum target number of fuzzing iterations against a field in a command can be determined using the following formula.

Minumum Number of target iterations for a field (Nf) = (# of unique valu for field generated by fuzzing library) X Multiplier Quotient

Multiplier Quotient = Field Fuzzing Priority (1 -10) X 1000

Field Fuzzing Priority is decided based on the role the field plays in overall operation of the command. (10 being most important, 1 being mostly untouched as per protocol spec)

The metric looks like the following one shown in Table 4.

Protocol Name	Command Name	Field Name	Number of iterations (Target)	Number of iterations (Current)
MS-SMB	SMB_COM_READ	Name of the field in the command being fuzzed: e.g ReadOffset	50000	30000
….	…	..	…	….

Table 4: Example Field Coverage Metrics

Similar to command sequences, it is also important that a combination of fields is fuzzed in a single iteration. This type of fuzzing attacks two different aspects of the validation routine at once. Many fuzzers already have this functionality but they randomly pick-up field combinations. Hence, we create a metric that lists field combinations based on documented protocol behavior. The combination is picked from the negative scenarios (where protocol specification says, must match/must be less than) and from the scenarios that exercise ambiguity in the protocol specifications. The metric looks like the one shown in Table 5.

Protocol Name	Command Name	Field Combination	Number of iterations (Target)	Number of iterations (Current)
MS-SMB	SMB_COM_READ	Name of the field in the command being fuzzed: e.g ReadOffset	200	150
….	…	..	…	….

Table 5: Field Coverage Metrics : Field Combinations

3.2 Scenario Coverage

The next set of metrics focuses on exploring various scenarios in which the product can be deployed. . It is important that fuzzing setup covers all of these scenarios since each of them can constitute an attack vector.

The metrics created in this phase are specific to the scenarios supported by the protocol being fuzzed. However, they can be broadly categorized in two types of metrics.

3.2.1 Software Settings

This metric captures the fuzzing setup details purely in terms of applied software settings from among the supported ones. Different settings internally translate to different code paths being covered and are important to increase coverage in black-box fuzzing approach. For example, while evaluating SMB we realized message compounding and fragmentation results in unique capture for a given operation. Similarly, playing around with other configurable settings results in changes in protocol behavior. The metric looks like the one shown in Table 6.

Protocol Name	Transport Layer	Fragmentation (Y/N)	Compounding (Y/N)	ASCII or Unicode	Fault Simulation	Stress (Y/N)
MS-SMB	This column shows the type of transport layer e.g.TCP, UDP	Shows if Packet fragmentation is turned ON or OFF	Shows if compounding is turned ON or OFF	Shows if ASCII or Unicode names are being used	Shows if network faults are simulated	Shows if stress mode is used
….	…	..	…	…	…	….

Table 6: Scenario Coverage Metrics: Software Settings

3.2.2 Hardware Settings

Usually, the protocol behavior changes when the underlying operating system or the hardware it is executing on changes. This behavior change is mostly due to the change in implementation details specific to a hardware/OS. This may expose vulnerabilities specific to an implementation. This also may help us to discover a bug easily which may be non-trivial to hit on other hardware/OS Matrix.
The metric looks like the one shown in Table 7.

Protocol Name	OS Version	Processor Architecture	Number of Processors	RAM
MS-SMB	OS on the machine e.g. Windows 7 sp1	Processor architecture of fuzzing target (x86 or x64 or ARM)	Number of processors on the target	RAM on target machine e.g. 1G
….	…	..	…	…

Table 7: Scenario Coverage Metrics: Hardware settings

3.3 Code Coverage and Bug Analysis

Our next set of metrics, are focused on white-box approach of fuzzing. The ultimate goal of code coverage metrics is to ensure that critical code blocks are covered during fuzzing. Besides this, the code coverage metric also serves as a useful method for providing feedback to the fuzzing tools on the set of input values to choose in order to cover specific code blocks thoroughly. With this method, instead of targeting entire code base for 100% coverage, we only target coverage in the prioritized list of code blocks which come out through code analysis. This prioritization is done by identifying the functions performing certain operations.

Functions constituting the attack surface which deal with receiving and sending data, handling network events and other low network operations are rated as first priority. Next, auxiliary functions providing services that require reading the untrusted data e.g parsing routines, validation routines (see Table 8) are given the second priority.

Using these metrics we continually enhanced our fuzzing tools to achieve more coverage of these code blocks.

Function Type	Function Name	Specific Command	Block-coverage	Fuzzing Pointers
Interface Routines	Function which operate at trust boundary level	If a specific command needs to be sent, enter here	Coverage in %	Feedback for fuzzing tools
Packet Validation Routines	Name of the functions which perform validation on packets			
Command specific Routines	Name of the function which performs command specific operation			
Helper Routines	Auxiliary functions that provide services that require reading the untrusted data			

Table 8: Code-Coverage Metrics

Bug analysis helps in finding patterns of possible security vulnerabilities. If vulnerability exists in a function, it is possible for it to exist in other functions performing a similar task as well. Besides this, bug analysis also helps us identify if the same vulnerability can be exposed from a different attack vector. This information can prove useful in updating the fuzzing tools to add more logic into fuzzing. Table 9 shows the metric used when doing bug analysis.

Bug Number	Root Cause	Action Items
333333	Failing to handle incorrect buffer lengths. Leaking memory in failure path	Fuzz buffer lengths
…..	….	….

Table 9: Example Bug Analysis Metric

3.4 Tools and techniques

Once we have metrics created for both black-box and white-box, we then evaluate the tools available to perform fuzzing. Ideally, the goal is to start with tools that cover both the popular fuzzing approaches, MITM and generational since each of these approaches has it's own benefit. For example, MITM fuzzing approach has the capability to attack a system after it has been brought into a specific state. Similarly, generational fuzzing approach gives users more flexibly in generating fuzzed packets.

Evaluation of generated values is also performed to ensure that a combination of both valid and invalid values are covered for a given data type. This is to ensure that the input space coverage is effective.

During fuzzing execution if parts of the metrics, whether black-box or white-box, show room for improvement in terms of coverage, these tools are enhanced to cover missing areas. Hence, by continuously improving the effectiveness of the fuzzing tools we stay away from getting saturated in our fuzzing efforts early in the fuzzing period.

The metric shown in Table10 explains the requirements for fuzzing tools and their development is tracked during the fuzzing cycle.

Tool Name	Technique Name	Protocol List (Target)	Protocol List (Current)	Missing Features
Tool1	MITM	MS-SMB MS-RAP MS-BRWS	MS-SMB	Capability to fuzz MS-RAP commands
….	….	..	…	…

Table 10: Example Fuzzing Tools and Techniques Metrics

4. Continuous Improvement

After successfully creating metrics that cover various parts of the protocol, we are then left with the task of executing them to continuously improve our fuzzing. This task is equally vital to ensure we keep penetrating into the different layers of protocol. Figure 2 demonstrates the interconnection among different parts of the fuzzing cycle.

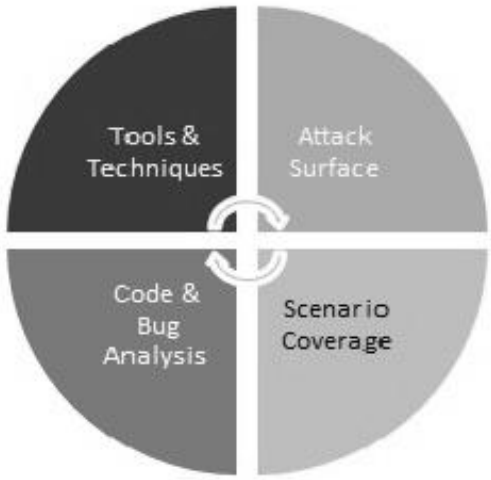


Figure 2: Continuous fuzzing improvement

In most cases, by analyzing the metrics of an attack surface, a tester gets a better insight on what additional scenarios should be deployed to cover any missing parts. For example, certain network components send commands based on a specific scenario. Unless that scenario is being exercised, metrics for those commands will appear zero. This leads to missing scenarios being added. The strong interconnection between attack surface and scenarios leads one to identify gaps between what's covered and what needs to be covered.

Filling these gaps results in improved code coverage as more code blocks get targeted. Additionally, code coverage analysis is performed which helps in providing feedback to the tools. Tools and techinques metric records this feedback and accordingly tools are enhanced continuously. This in turn results in better coverage, and the fuzzing becomes more and more effective with each feedback cycle.

5. Metrics in Planning

Different parts of the metrics are created at different stages of planning process. The planning process can be divided in four major stages:

Stage1 - Scope the fuzzing effort: In this stage we freeze on in-scope protocol list. This allows us to identify the attack surface and create relevant set of metrics under Attack Surface.

Stage2 - Prioritize product usage scenarios: Once the protocol list is identified, we plan and prioritize scenarios under which the product is exercised. This stage enables us to create Scenario Metrics.

Stage3 - Perform code analysis: When planning targeted fuzzing for specific code sections using whitebox approach, this stage helps us create Code Coverage and Bug Analysis Metrics.

Stage 4 - Identifying tools and techniques: In this stage we focuse on tools planning for the fuzzing project. We create Tools and Techniques metric to evaluate and improve existing tools and develop new ones.

Figure 3 describes how each planning phase helps in creating a specific metric set.

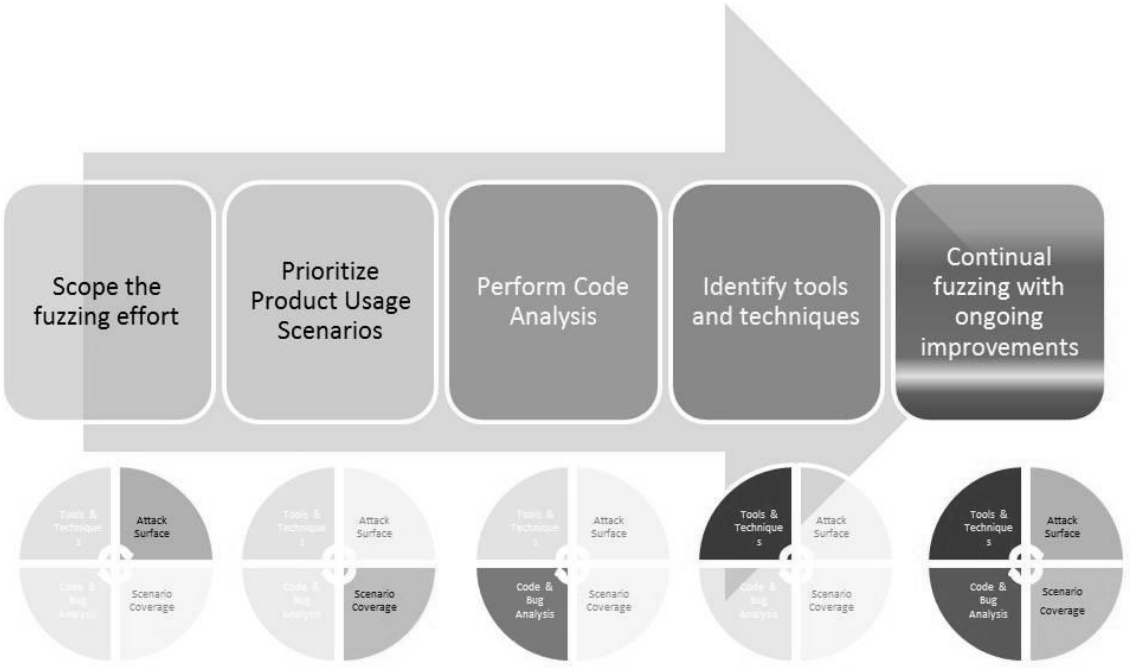


Figure 3: Metric creation in planning process

6. Case Studies & Results

We used Metric-driven fuzzing strategy to fuzz three network protocols namely MS-SMB/MS-SMB2, NFSv4.1[6], iSCSI[7]. This structured approach helped us get a high coverage in each of the metric groups.

Attack Surface: Table 11 displays results achieved under protocol coverage for different protocols. While fuzzing MS-SMB protocol, we also included related protocols, which use SMB as transport e.g. MS-RPC, MS-RAP. Together there were 142 commands for which we needed coverage. Around 36 commands among these are either deprecated or obsolete in Windows 7. These are the commands that are specific to older dialects and are not generated by current Windows SMB client but still supported by current Windows SMB Server. Similarly certain subcommands were difficult to generate. In order to get coverage to 100% multiple tweaks were done to the fuzzing tools like forcing client to advertise lower dialects only etc.

Protocol	Total Commands	Commands Covered
MS-SMB and Related	142	100%
NFSv4.1	84	95%
iSCSI	38	79%

Table 11: Results of metric-driven fuzzing: Protocol coverage

Scenario Coverage: Table 12 displays results achieved under scenario coverage. For MS-SMB 10 unique scenarios were identified. Since MS-NFS and MS-iSCSI were fuzzed only for latest versions of operating system, number of scenarios was much less compared to MS-SMB. However, each of the identified scenarios, added to the code coverage achieved just from the default configuration. (Fresh installed Windows Server 2008 R2 on amd64 architecture).

Protocol	#of unique Scenarios	Scenarios Covered
MS-SMB and Related	10	100%
NFSv4.1	5	100%
iSCSI	3	100%

Table 12: Results of metric-driven fuzzing: Scenario coverage

Code and bug analysis: As presented in Table 13, this effort for SMB resulted in 170 functions being identified as target for fuzzing. While initial code-coverage in these functions during fuzzing was surprisingly low, improvements in tools and adding scenarios led to increase in code coverage up to 90%. This effort for NFS and iSCSI is planned for future.

Protocol	#of functions shortlisted	Functions Covered
MS-SMB and Related	170	90%
NFSv4.1	Planned	--
iSCSI	Planned	--

Table 13: Results of metric-driven fuzzing: Code coverage

Tools and Techniques: Table 14 shows how each of the fuzzing approaches were able to hit majority of the commands for each of the three protocols. In certain cases generational fuzzing was difficult to achieve in all the scenarios (e.g. multiple clients sending packets over UDP like in MS-BRWS). That explains the lower numbers for the generational fuzzing in some cases. MITM approach was continuously validated against the penetration of the fuzzed packet to make sure the packet is not rendered invalid and we continue to achieve high code coverage for all the commands it supports.

Protocol	Generational Fuzzing	MITM fuzzing
MS-SMB and Related	80%	100%
NFSv4.1	100%	100%
iSCSI	100%	100%

Table 14: Results of metric-driven fuzzing: Tools and Techniques

In addition to the coverage in different metric groups, this approach helped us find more than 50 security issues in these 3 protocols. While some of the bugs were easily discovered in the initial stages of fuzzing, we found majority of the bugs when the tools became more effective due to the feedback from code coverage and scenario coverage metrics. All the bugs we found made into bug analysis metric, which helped in dynamic tuning of fuzzing to find variants of the initial issue.

7. Conclusion

Network protocols are complex systems with a large attack surface in multiple usage scenarios. During a fuzzing effort, one cannot get an insight of the various coverages being achieved, without measuring fuzzing using appropriate metrics. This leaves the fuzzing effort incomplete and less effective as the fuzzing process misses out on the opportunity to continually improve itself based on the collected data.

The most important part in fuzzing execution is procuring the right set of metrics that covers various parts of the protocol and then using them as a criteria for fuzzing completeness. When done appropriately these metrics provide numerous benefits to industry practitioners. First, meaningful targets can be set based on the different protocol aspects such as related protocol, command, field coverage and different fuzzing techniques. These targets then guide towards a planned exit criteria with thoroughness. Secondly, quantitative analysis can be performed on the various coverages such as protocol, code blocks, scenarios etc. and continuous improvements can be made in the tools to make them more effective.

This paper demonstrated the use of metrics that include both, the black-box approach and white-box approach to fuzzing. Black-box metrics are based on protocol specifications and usage scenarios. White-box metrics depend upon knowledge of the protocol implementation.

The case study demonstrated the application of metric-driven strategy to fuzzing using 3 industry-standard network protocols. It illustrated the potential of this strategy to help researchers in the network security field.

Bibliography

[1] Writing Secure Code by Michael Howard and David LeBlanc (Second Edition)

[2] Mikko Varpiola and Ari Takanen. How to Deploy Robustness testing.

[3] Fuzzing for Software Security Testing and Quality Assurance

[4] [MS-SMB]: Server Message Block (SMB) Protocol Specification

[5] [MS-RAP]: Remote Administration Protocol Specification

[6] Network File System (NFS) Version 4 Minor Version 1 Protocol

[7] Internet Small Computer Systems Interface (iSCSI)

On the Way to Meeting a Mandate: Transitioning to Large Scale Agile

Aashish Vaidya, Catherine Row, Mark Jackson
aashish.vaidya@cambiahealth, catherine.row@cambiahealth.com,
mark.jackson@cambiahealth.com

Abstract

Over the last decade, agile practices continued to gain a foothold in many companies. Many of the practices were first tried on a smaller scale, and tailored to green-field projects. But, as agile practices are adopted at multi-site, larger companies, with complex back office projects, they bring with them additional and unique challenges. Take for example, Cambia Health Solutions, a not-for-profit, health Solutions Company. Cambia has a relatively large IT group, which develops its own software solutions and integrates software from other vendors. It has a distributed workforce that spans four states with many work-at-home employees. Cambia's mainline business is regulated by various state entities and national agencies. Many of Cambia's projects are large, mandated compliance projects that have a fixed time box and a set scope. Missing either the deadline or scope could potentially mean losing a significant portion of its business or facing monetary penalties. Can agile practices work and scale to solve business problems facing the company? What if you also operate within an IT organization that is structured with functional silos? And what if you have reluctant business partners who define software requirements and perform final verification of software solutions? To tackle these challenges, Cambia organized an Enterprise Transition Community to lead the transition from a patchwork of agile practices to a large-scale agile implementation. Find out how we carried out this transition, what we have learned along the way, and how that might help organizations of all sizes take on a similar challenge.

Biography

Aashish Vaidya *is a Technology Manager leading Specialized Teams at Cambia Health Solutions. He is a founding member of Cambia's Enterprise Transition Community, and other Best Practices Exchanges. He also serves as an internal coach on Agile and QA practices. Aashish has over 20+ years working in technology development and leadership positions for companies such as Compaq, Intel, and Kronos Incorporation. In 2011, Aashish was a panelist on Technology Association of Oregon's panel discussion: QA's Role in Agile. He is a graduate of Texas A&M University, and holds a Bachelor of Science degree in Aerospace Engineering.*

Cathi Row *is an IT Talent Development Program Manager at Cambia Health Solutions. Cathi brings over 25+ years of experience in organizational development, software training, and transition management. She has been with Cambia for over 15 years and holds a Bachelor of Arts degree in Education from Fort Lewis College in Colorado.*

Mark Jackson *is a Director of Technology at Cambia Health Solutions, currently overseeing the implementation of the ICD-10 federally mandated program. He is a founding member of Cambia's Enterprise Transition Community. Mark has 20+ years of software development and leadership experience. Mark holds a Bachelor of Science degree in Computer Science from California State University, Chico.*

1. Introduction

Cambia Health Solutions is not-for-profit total health solution company dedicated to transforming the way people experience the health care system. Like many other health care related companies, Cambia faces constant market and internal organizational challenges.

Cambia, formerly known as The Regence Group, is a parent organization that comprises two groups of companies: Health Insurance Services (HIS) and Direct Health Solutions (DHS). Health Insurance Services is Cambia's mainline, regulated health insurance business. It was created as a result of the merger of four different Blue Cross Blue Shield entities from four states: Oregon, Utah, Idaho and Washington. The Direct Health Solutions group includes companies that provide various affiliated healthcare solutions.

The Cambia IT department is a shared services department, which provides software development, integration as well as IT infrastructure services to both HIS and DHS companies.

In order to better meet the external marketplace and internal delivery challenges, in July of 2011, Cambia IT went through a major department restructuring and initiated a large-scale effort to change from a traditional waterfall Software Development Life Cycle (SDLC) to Agile/Lean development methodology (Scrum/Kanban).

An Enterprise Transition Community was formed to coordinate and shepherd this large change management process across IT and other business units. This paper outlines how the Enterprise Transition Community has led the transition to large-scale Agile at Cambia.

2. Prior Organization Structure and Agile Practices

Prior to the reorganization in July of 2011, Cambia IT was organized into traditional functional silos: Architecture and Design, Software Development, Quality and Release, and Infrastructure. Enterprise Program Management Office (EPMO), which was outside this structure, provided traditional project management and business analysis services. Various business units also maintained their own business analyst and user acceptance testing (UAT) staff.

Over the last few years, various groups in the IT organization were increasingly adopting agile practices for development and delivery. However, project inception, the budgeting and business requirements gathering process, and the user acceptance testing before product deployment still followed traditional waterfall processes. Uneven application of agile processes created confusion between the IT department and other business divisions. This divide led to frequent issues with process and role confusion and delivery and transparency problems. Deliveries had quality problems; they missed customer requirements and deadlines. Frequently employees had to perform major heroics to meet business goals.

2.1. Workforce Makeup, Skills and Project Staffing

Using traditional project management practices, projects were initially staffed based on the resource availability of full-time, regular employees. As staffing for maintenance activities occupied a majority of the regular workforce, new projects were largely staffed with a contingent workforce.

To deliver large enterprise wide programs and fulfill its ongoing project needs, Cambia IT relied heavily on a contingent workforce. It wasn't uncommon to see the regular to contingent workforce ratio of 1:1 and in some cases it approached as high as 2:3.

Continual recruitment and retention of a large contingent workforce also meant software managers and development leads incurred additional contingent worker oversight responsibilities, further limiting their delivery efficiency and leaving them with little time to pursue any technical or process improvements.

Project work was estimated using a 3-6 month time horizon, and project suspensions and extensions were common occurrences. This required additional management oversight directed towards continual resource management. The constant churn on project suspensions and restarts meant constant forming and reforming of teams and loss of productivity. Project and maintenance activities caused tremendous strain on key personnel with certain rare skillsets.

Given that most of the project work involved new product development or integration of new technologies, the full time workforce working on maintenance activities found it difficult to acquire new technical skills. Additionally, balancing new development and maintenance work created unsustainable work pace, leading to employee dissatisfaction.

The transition from new development to maintenance wasn't always factored into project plans. Business justification generally trumped quality criteria and hence the support organization had to grapple with quality issues introduced during the project implementation.

2.2. Nascent Agile Efforts

Within IT, prior to the re-organization, there was an active grass roots group called the Agile Working Group that provided guidance on basic agile best practices. Cross-functional agile teams were formed and some of these teams even had embedded QA personnel. However, this agile movement was largely IT-centric and business partners still followed waterfall practices. Without active IT executive management support, it did not carry much weight in convincing EPMO, UAT, and other business groups to embrace agile techniques.

As a result, the IT agile development process remained sandwiched between Waterfall requirements gathering, design, and analysis work at the front end, and with UAT at the tail end of the delivery cycle. The hybrid approach worked in small pockets and had some limited success - when the IT and business stakeholders were more aligned and willing to experiment and collaborate. Unfortunately, the hybrid approach put a strain on the EPMO project governance structure that was oriented towards traditional waterfall practices.

Within IT, project teams were constantly forming and storming, due to heavy reliance on contingent workforce. This made it very hard for agile practices to take firm root and contribute positively towards software deliveries.

2.3. Larger Projects and Agile

On large projects, which required multiple IT teams, the process confusion was further magnified. Some IT teams used agile practices; others did not. This meant that Business Analysts had no common way to communicate project requirements to the IT team. Additionally, because IT teams were aligned to software subsystems, when a change in a single subsystem needed to be made, its impact to multiple IT teams and their stakeholders, weren’t always well-understood or well-communicated.

The lack of a single consistent development process made it much more difficult for EPMO to create and track budgets, resources and project plans across software delivery teams. For the UAT group, different scheduling and delivery mechanisms created difficulties in coordinating and sourcing business testers in a timely fashion. Even within IT, process confusion was common. The deployment teams struggled with two sets of rules: one for those practicing Agile and ones that followed Waterfall staged-gate processes. Team members changing teams or projects would encounter a patchwork of practices from one team to

next, leading to higher ramp-up time and lost productivity. All this contributed to the mistrust between various departments and further increased the divide between IT and other business units.

Additionally, even as teams were experimenting with Agile and learning newer and better ways of doing things, these gains were localized and insulated from others in the company. There was no mechanism to spread these patterns and practices across the organization. Moreover, due to constant schedule pressures, it was difficult for agile teams to experiment and incorporate agile engineering practices, stunting their maturity.

3. Cambia IT Reorganization

The ability to deliver large multi-team, multi-site, mandated programs, with high quality, on time and on budget, still largely remained elusive. In July of 2011, the new CIO and his IT executive team undertook major structural changes in order to meet constantly changing business needs, delivery predictability and high quality.

Dean Leffingwell and many other agilists list the makeup of the organization along functional lines, instead of product or business application lines, as a major impediment to adoption of agile (Leffingwell, 2007, chap. 8, loc 2168 of 7271). Accordingly, Cambia's IT organization, which was composed of functional silos of architecture and design, analysis, development, quality and release divisions, was restructured. The new divisions at CIO staff level were aligned to provide services to corresponding business units. In addition, the decision was made to fully adopt agile practices across the delivery lifecycle.

To support this large cultural shift, a Talent Development Program Coordinator position was created and filled. And in the fourth quarter of 2011, best practices communities were launched to provide a means for functional groups to share best practices.

However, the best practices communities were focused on their functional areas. There wasn't an overall, coordinated process to rollout Agile across the IT organization and to business partners. In December 2011, the Enterprise Transition Community (ETC) was formed to help with change management and transition process to large-scale agile. The purpose of the ETC: provide a focus, coordinated effort to deal with the thornier issues of scaling agile to deliver large-scale, multi-team programs; to adopt agile practices throughout the entire SDLC; and to eliminate process, role and tool confusion. The ETC was tasked to pilot this transformational process on ICD-10, a large, multi-team, federally mandated program. ICD-10, which stands for the International Classification of Diseases, 10th revision, has a compliance deadline of October 2013. Within Cambia, the program is far-reaching and involves half of the software delivery and support teams in IT, and affects almost all business units in the company.

3.1. High Level Timeline of Reorganization

Here is a high level timeline for the Reorganization:

- May-June 2011 – New CIO and CTO hired.
- July 2011 – IT reorganized from functional silo based units to cross-functional divisions aligned to business units.
- August 2011 – Change initiatives launched; Software Quality Best Practices Exchanges (SQBPE) launched; Talent Management Program Manager position created.
- October 2011 – Talent Management position filled; Agile Working Group morphed to Agile Best Practices Exchange (ABPE).
- December 2011 – ETC was formalized; Companywide training initiative started.
- January 2012 – ETC tasked with leading change to Agile and ICD-10 chosen as the large-scale pilot program.

- January 2012 – First multi-team release planning session for ICD-10 program.
- March 2012 – Portfolio planning for ICD-10 introduced.

3.2. Agile and Shared Teams

After the reorganization, a cross-functional agile team, consisting of Systems Engineers, Systems Analysts, Developers, Testers, and Test Automation Engineers, became the fundamental building block for organizing work. Leffingwell calls this "the fractal unit of agile" (Leffingwell 2007, chap 9, loc 2277 of 7271). The IT department is organized into nine CIO staff level divisions, of which five have responsibility for software development. These software development related divisions are further split into more than 30 different agile teams. In addition to dedicated agile teams, a few shared services teams were also created. These shared services teams included a small number of employees who have specialized skills which couldn't be distributed evenly within each IT division. These new shared services teams include:

- Specialized Skills Team: Provides performance testing, data validation, test data creation and QA test automation.
- Quality Focus Team: Provides services to promote, improve, share and implement practices that focus on building quality up front.
- Data Modeling Team: Provides data modeling work and guidance to agile teams.

The agile teams and shared services teams represent more than half of the 600 employees within IT organization. Another 250 team members are part of the infrastructure teams that provide basic IT functions including desktop and help desk support, release management, hosting, production deployment and monitoring, and other support functions. The rest of the organization consists of IT management.

Currently, only select shared services teams and infrastructure teams follow agile practices. More teams are slated for internal assessment and transition to agile practices during the latter half of 2012.

3.3. Enterprise Transition and Other Communities

For large scale agile changes, IT executives and other leaders knew that grassroots efforts, though effective on selected teams, would not scale systematically across and into the enterprise. In order for Agile to gain wider acceptance and overcome resistance within IT and with business partners, a consistent and focused change management effort was needed.

A coordinated community with passionate individuals working towards a common goal was required, since the patchwork of Agile implementation wasn't effective. The Enterprise Transition Community was formed to lead and manage the transition and adoption to Agile. This community also needed the explicit backing of IT executives, so that they could provide "air cover" to augment the energy and drive of grass roots efforts. Both, Ken Schwaber (Schwaber 2007, pg. 9) and Mike Cohn (Cohn 2009, pg. 63), consider an Enterprise Transition team a keystone community that "initiates, encourages and supports an organization's" transition to Agile. Both, Schwaber and Cohn recommend that the ETC be composed of the highest level executives from IT and other business partners.

However, Cambia's ETC does not derive its core membership from senior leadership, but instead from leaders from IT divisions, EMPO and select business units. At Cambia, the ETC coordinates with other role-based communities and Best Practices Exchanges (BPEs) to roll out Agile across the enterprise.

The ETC is chartered to drive changes at team, program, and at portfolio levels. Generally, in major rollout situations such as this, textbook Agile would suggest trying new ideas on smaller, less risky projects. However, in a departure from standard practices, the ETC was asked to use ICD-10 as a pilot program. Cambia IT's senior leadership felt that trying out newer practices on ICD-10 was the best way

for Agile practices to take hold and for the organization to develop institutional “know-how” of handling large complex programs.

The ICD-10 program is akin to the Y2K remediation software effort, as the ICD codes are central to Cambia’s business. A major change to the ICD code set format involves nearly every business units within Cambia. It requires use of 15 of the 33 software delivery teams, software suppliers, and it requires coordination with external business partners such as Blue Cross Blue Shield Association plan members, commercial service providers and medical partners.

In addition to the ETC, other communities were formed to target functional groups or roles:

- Agile Best Practices Exchange (ABPE) – Umbrella exchange for all things Agile.
- Software Quality Best Practices Exchange (SQBPE) – Forum for software quality engineers to create, share, and improve software quality best practices.
- Agile Coaching Community – Internal agile coaches coordinating with the ABPE and the ETC and other groups to provide Agile coaching support.
- Other groups such as Agile Training Community, Information Security Community, Product Owner Community.

These exchanges work in coordination with the ETC to help in the transition process.

3.4. Staffing Mix and Talent Development Program Manager

The IT Executive Team made another crucial and much needed structural change. The CIO staff worked with the Finance team to change the staffing and budgeting rules, which were skewed towards hiring of contingent workers for projects and regular workforce for maintenance work. The CIO staff wanted to make this change for few reasons: provide more opportunity for regular workforce to learn newer technology needed on projects; send a message that regular workforce is more valued than the contingent workforce; and, allow for retain knowledge of agile practices as opposed to having it walk away when the contingent worker leaves.

This led to an IT initiative to reduce contingent workforce relative to regular IT staff. An initial goal of approaching regular employees to contingent workforce was set to 4:1 ratio within two quarters**.** This target was achieved ahead of schedule.

The IT Talent Development Program Manager position was also created to assist the IT Executive Management and the Enterprise Transition Community in effecting the large scale transition and change management process. The position was also tasked with nurturing and formalizing best practices communities, and creating a strategy to drive learning and development throughout the organization.

4. ETC Initiated Practices

From January to June 2012, in coordination with IT, business leaders, and with other communities, the ETC has launched several coordinated initiatives to improve agile practices. These include: helping in creation and support of self-organizing agile teams; training; eliminating agile process, role and tool confusion; and, adopting agile practices throughout the software development lifecycle. The ETC is expected to be operational for additional two to three quarters to complete the transition process.

Through the ICD-10 project, the ETC is also developing program and portfolio level practices. After these practices are piloted on the ICD-10 programs, other programs are using these patterns to organize their own deliveries. The ICD-10 program, which has been operational since early 2011, and was considered off-track, has benefitted from ETC initiated practices in first two quarters of 2012. It is now back on-track and is better positioned to deal with schedule changes

4.1. Organization Wide Agile Training Plan

In the past, various groups arranged their own agile training plans. These piece-meal training plans were tactical and overlooked many people seeking the correct type of training. For example, Cambia IT has over 70 team members who have gone through certified Scrum Master training. However, only 15 of these trained Scrum Masters are leading teams, leaving the other 18 agile teams with Scrum Masters who have not received formal training. Additionally, the training plans did not include EPMO, UAT and other business partners.

The ETC was able to craft and fund a holistic training plan. The initial training plan included three different categories targeting different audiences:

- Basic Agile Overview – Full day training for all team members and anyone with general interest in learning about Agile.
- Product Owner Training – Two full day training targeted to product owners.
- Agile Overview Training for Managers – Half day training geared toward managers with team members participating in agile activities or any managers interested in learning about Agile.

These training sessions were designed to include team members from various IT and other groups. This has allowed for much livelier and productive interaction amongst the various groups during and after the training sessions.

During the initial training sessions, which were delivered by outside consultants, Cambia also developed a cadre of internal trainers through a train-the-trainer approach. These internal trainers are currently able to offer these courses regularly, allowing IT to stretch its training budget.

Apart from formal training, several BPEs hold educational events to supplement training. Some have taken on the task of developing self-service educational materials for employees.

4.2. Program and Portfolio Level Practices

The ETC also initiated practices that transitioned Cambia's agile practices from a patchwork to a more large-scale, enterprise-wide Agile, and introduced program and portfolio level practices such as:

- Enterprise wide synchronized sprint schedule (Cohn 2009).
- Multi-team Release Planning.
- Scrum of Scrum meetings (Schwaber 2007).
- Hardening Sprints, Architectural Runways, (Leffingwell 2012).
- Use of Portfolio Management to coordinate and track epics and themes at program level.

The new practices have been successful on the ICD-10 program, and plans are underway to roll these out to more programs. Discussion on these practices is out of scope for this paper; however, next year, we hope to make a complete report in this area.

5. Measuring the Progress

As ETC continues to shepherd the transition process, it has also developed ways to measure the progress of this transformation.

5.1. Agile Team Composition Scorecard

One set of measures is called Agile Team Composition Scorecard, illustrated with the following table, (Note: The table is truncated and modified from the original Scorecard used internally at Cambia)

Team Name	Manager	Scrum Master	Product Owner	Ratio of QA to Dev	SE / Architect	Team Co-Location Index	Size	Team Health Score
Team 1	Manager A	y	y	0.2	y	8	9	5
Team 2	Manager A	y	y	1	y	8	5	5
Team 3	Manager B	y	y	0.125	y	9	11	4
Team 4	Manager B	y	y	0.231	y	10	17	3
Team 16	Manager L	y	y	0.2	y	7	8	2
Team 17	Manager L	y	y	0.143	y	8	9	4
Team 18	Manager M	y	y	0	y	7	5	4
Team 19	Manager M	y	y	0.5	y	9	4	5
Team 20	Manager N	y	y	0.111	y	8	14	5
Team 29	Manager T	y	n	0.25	y	6	8	4
Team 30	Manager U	y	n	0.4	y	6	7	4

Table legend:

Green	Yellow	Red

The measures on the Scorecard are derived from basic Scrum practices. The Scorecard uses traffic light colors. Below is an explanation of these measures.

Scrum Master:

- Each team must have a Scrum Master.
- Criteria: Green - if each team has a Scrum Master; otherwise red. Additional guidelines not captured on the score card: Scrum Masters are typically expected to spend 60-70% of their time towards the role; and they should serve on no more than 2 teams.

Product Owner:

- Each team must have a Product Owner.
- Criteria: Green - if each team has a Product Owner; otherwise red. Additional guidelines not captured on the score card: Product Owners are typically expected to spend 50% or more their time towards the role; and they should serve on no more than 2 teams.

Size:

- This measure tracks the size of the team. We are aiming for the proverbial “two pizza team.” (Cohn 2009, pg.177).
- Criteria: Green – for team size of 5-9, not counting Scrum Master and Product Owner; otherwise red.

QA:Dev Ratio:

- Set at 1:4. The CTO has set the QA:Dev ratio target to balance the functional skills on the team. The CTO feels the ratio should be even higher, to deliver high quality software that delights the customers, and lessens the burden on the UAT and business testers.
- Criteria: Green – greater than or equal to 0.250 (1:4); Yellow – greater than or equal to 0.143 (1:7) and less than 0.250 (1:4); Red if less than 0.143 (1:7).

SE/Architect:

- Over the two quarters, a Systems Engineer or an Architect is assigned to support 3-4 teams. Teams normally consult with their Architects for technical direction and for resolving cross-team issues, and seeking guidance on stories and epics during planning sessions.
- Criteria: If a team has a "named" assigned Architect for their team, they receive a green; otherwise red.

Team Co-location Index: Cambia is already a distributed company operating within 4 states in various locations throughout these states. As a cost-saving measure, prior to the re-organization, there was a corporate wide push to make employees become work-at-home, remote workers. Over time, though, the preference is to have more co-located teams. For now, the Co-Location Index provides visibility on the degree of distributiveness, on a team by team basis:

- ((Total team members less # of unique team member locations less one divided by the total number of team members) multiplied by 10
- An entirely co-located team receives a score of 10; completely distributed team receives a score of 0.
- Criteria:
    - Red - less than 4.
    - Yellow - less than 8, greater than 4
    - Green - greater than or equal to 8
- Team with office workers at a single site, even if they are spread out in different buildings are counted as co-located (Since the index was created, team members spread across various buildings are now consolidated at common team sites – called "agile towns"). Each remote worker is counted as occupying 1 location.

Combined, these six measures define the nature of the team - the "fractal unit of agile". In the section on Takeaways, below, we discuss how the Scorecard has helped Cambia make an agile team, the fundamental work unit.

5.2. Other Metrics

In addition to the Agile Team Composition Scorecard, the ETC is also developing measures along three broad categories:

- IT Mission metrics - to measure whether IT is fulfilling its mission providing services to other business units. These metrics are constructed along the lines of customer value, quality, on-time delivery and budget. Currently these metrics are tracked by our release team and in our agile tracking software. We are making improvements to get them tracked on a per team basis, similar to the Agile Team Composition Scorecard. This way, the teams can use the metrics to make continuous improvements.
- Agile Practice metrics - to track adoption of agile practices and maturation. Many of the measures in this area are getting introduced iteratively. The ETC is putting in place a mechanism to track an initial set of measures: committed to completed story points; percentage of stories with acceptance criteria; "formal" sprint review with business stakeholders; team retrospectives.
- Business Values metrics – to measure how well the agile teams along with their partners are delivering business value through the use of Agile methodologies.

6. Takeaways

Transformation of Cambia's transition to large scale agile is still a work in progress. However, in a year's time, much progress has been made and there are important takeaways and lessons learned from this process. An organization generally faces two sets of hurdles – those that stem from company's culture

and structure; and, those that stem from the challenges of scaling agile practices to work for large projects and programs. The following subsections describe how these are playing out at Cambia.

6.1. Enterprise Level Community and Change Management

Having an enterprise level community focused on the change management process has been crucial in this large-scale transition process. At Cambia, we have used change management models to aid in understanding and leading this type of transition. One such model is outlined in the book *Influencer: The Power to Change Anything* (Patterson, et.al. 2007) This model consists of two domains: motivation ("will it be worth it") and ability ("can I do what's required"), which is then further subdivided into personal, social, and structural sources. This model is illustrated with the following diagram:



As a transition community, the ETC taps into these six sources of influence to drive change and overcome the natural resistance encountered during the large scale transformation process.

6.2. Communities Reflect their Constituents

The core groups on ETC as well as the other Cambia communities (such the Agile Best Practices Exchange, the Coaching Community and the internal Agile Trainers Community) all have derived their memberships from the entire enterprise. Constructing cross-functional teams has proven very effective in influencing and building creditability with non-IT business units. For example, having ETC members from business units allowed us to fill the product owner gaps on teams. We think this closer collaboration of business involvement with the agile teams has been very impactful and will continue to pay big dividends in the future.

As each group has a stake in seeing the success of this transition, inclusion of team members from various groups allows each to have a voice in this process. From a change management perspective, the membership of these communities taps effectively into social motivation as a source of influence.

6.3. "Scrum is not a Synonym of Agile"

Though, many agile purists may feel that choosing a particular agile method stifles innovation on teams, Scrum as method of choice has so far proven very effective for Cambia. This decision to standardize on Scrum has allowed team members, and business partners to learn a single method together, and be conversant with it. This has allowed the organization to largely avoid unnecessary "method wars", which

usually confuse and shut out more skeptical business partners. Scrum has provided the non-IT groups with a single, consistent interface with IT and vice versa.

That is not to say that there isn't any variation in practices. When it is warranted, Agile coaches have judicially introduced Kanban to a handful of infrastructure teams. The guiding principle for ETC and other communities continue to be: encourage practices that are more suited to each team and enable it to become more effective.

6.4. "Do as I Say and Do As I Do"

Most of the communities, which are not software development teams, have now adopted agile practices for organizing their own work. Most of them maintain a product backlog, write stories, develop acceptance criteria, and conduct retrospectives. Many communities practice daily standups, create release plans and roadmaps.

This has been advantageous in a couple of ways. It enables ETC team members who lead the transition to develop an appreciation of the challenges and triumphs that software delivery teams and stakeholders face. Another benefit of the individual adaptation of Agile is it sends a message to all team members that "we are all in this together."

The use of Agile in ETC and other communities also means that changes are rolled out at a measured pace, instead of in a torrent. Each community takes up only as much work as they are able to complete in a given Sprint. This allows time to develop consensus and incorporate any feedback. This largely avoids the "top-down", command-and-control approach that generally meets heavier resistance and doesn't stress people out as much.

6.5. "Where's my Tribe?"

With the creation of cross-functional teams, members of the former functional organizations feared that they would lose their respective "tribal" connections. This was especially evident in many of the software QA Engineers. They felt that the emphasis on quality would diminish even further, as there would no longer be a collective voice speaking for their concerns. Fortunately, the formation of best practices communities, such as the SQ BPE, has allowed QA Engineers and members of other disciplines, to feel connected to their former "tribes". BPEs serve as conduits for team members from former functional silos in maintaining cohesiveness in their practice, and establishing and sharing new practices. It also helps team members in cultivating a stronger voice of for their function in their own agile teams.

Best Practices Exchanges have also allowed for new "interest" groups to develop outside of the functional skills group. An example is Secure Coding Practices Community. The formation of this community and its activities has created greater awareness and traction on software and infrastructure security concerns.

6.6. "You Mean You Don't Have Time"

One of the bigger challenges has been obtaining consistent time commitment from team members allocated for work on the BPEs. The norm in most of the communities is for each core team member to allocate 10 percent or more of their time to BPE activities. Each BPE typically also asks for a commitment for three months. This has proven to be quite challenging for the team members as well as their managers. This slows down the pace of deliverables that the communities would like to actually produce from their activities.

For example, the agile coaches are not dedicated to coaching tasks alone. Many participate on other best practices communities, along with their main work related duties. Ideally, the ETC would prefer to have agile coaches assigned to support agile teams, much like we do with Architects. However,

availability of coaches does not keep up with the need, which in turn contributes to slow adoption of agile practice refinement and maturation on many teams.

6.7. Training Helps Lower Resistance to Change

Targeted and recurring training sessions have helped lower some of the resistance to change. Mixing attendees from various groups has also helped in chipping away at some of the barriers between previously adversarial groups, who may have been at odds with each other due to difficult projects and their experience with Agile.

Providing training to managers (via the Agile Overview for Managers) has also proven to be successful. As it happens many times, management can easily thwart team members, who are more receptive to change to Agile. Following the training, we were able to close the product owner gaps on many teams, as the corresponding business unit managers allowed their reports to take on the product owner role. Also, the managers are more supportive of business testers who are collaborating with the agile teams.

Training also taps into one of the fear that many team members face: “Do I have the ability to do this new thing?” Along with formal training plans, many Best Practice Exchanges have put targeted learning sessions together.

Formal training and other learning opportunities have also sent a powerful message to many regular employees, who felt the company in the past had under-invested in their career development.

6.8. “My Customer Won’t Talk to Me”

It is probably ironic and perhaps hard to believe that in IT organizations, such as Cambia, where customers are sometimes literally down the hall (as opposed to say, a shrink wrap software or an SaaS offering), many software delivery teams found it hard to engage customers in providing ongoing input and feedback during development process.

The use of Agile Team Composition Scorecard, which is reviewed with executives every sprint, provided the visibility and impetus to fill the product owner gaps for teams. Also, as one of its deliverables, the ETC produced a checklist for business stakeholders to secure a product owner for each team. Cambia started the year with close to 70% of the teams missing a “named, dedicated” product owner, but by the end of second quarter, more than 90% of teams now have one. What’s also encouraging to see is that many Product Owners have chosen to co-locate with their teams.

With better definition of the team structure within IT, the UAT group has assigned Test Coordinators and Subject Matter Experts to each of the agile teams. The continuing engagement of both the Product Owners and the testers from the business has allowed for ongoing input from the business during the development cycle. The use of Agile Scorecard has helped addressed one of the main issues that faced past agile implementation – the lukewarm engagement from the customers.

6.9. “When Data Talks, Opinions Walk”

Cambia’s CIO has a saying, “when data talks, opinions walk”. As the Agile Team Composition Scorecard provides the visibility and means to close the product owner gap on the agile teams, it is similarly shedding light on other measures as well. The Scorecard has allowed identification of gaps in roles, and facilitated proper makeup of the team.

For example, IT Executives have questioned hiring practices of IT Managers whose teams might have a lower than a 1:4 QA:Dev ratio. It should be noted that though we have made improvements in this area,

they are not as dramatic as Scrum Master or the Product Owner roles. We anticipate this measure to take longer to remediate.

The Scorecard has put subtle, but powerful pressure on many IT managers who may be resisting the change. No IT manager likes to see his or her team's score show up "red" for too long. There have been spirited discussions on whether the Scorecard measures make any sense or if they are effective. However, over time, as team managers have come to realize that the Scorecard isn't used as a punitive tool, they have been more willing to use the information from it to make changes. Many managers with teams that are too small or too big are looking for logical ways to consolidate or split their teams and backlogs. Many managers have used this information to adjust their hiring plans. Those who do not have open requisitions are looking for cross-training opportunities within the teams. Those teams with a low Co-Location score have also used it for travel budgeting so that they can co-locate teams for planning sessions.

7. Conclusion

Changing from the traditional waterfall method to Agile presents daunting challenges to many organizations, as this transition exposes existing, but largely ignored fundamental structural problems. In many instances, uneven application of agile adoption leads to other issues as organizations learn to deliver code faster and at set intervals, but unfortunately at the expense of software quality. Organizations move faster initially, only to slow down later under the weight of technical debt.

Scaling agile practices has been a challenging process at Cambia, and many challenges still remain as we continue to evolve these practices at the enterprise level. Delivering on fixed duration, fixed scope, and federally mandated projects may seem like an incompatible use of Agile. However, through the formation of the Enterprise Transition and other communities, Cambia is benefiting from a concerted effort to transition to Agile.

Along the way, we are discovering that driving transition activities through self-managed communities, and eschewing heavy compliance or governance structures, seems to lower the initial resistance and has made the organization much more amenable to adoption large-scale agile practices. Our efforts have also shown that use of effective measures such as Agile Team Composition Scorecard can lead to desired results. We think many of the things we have learned can apply to organization of any size or type, which is looking to change or is in the process of changing to Agile.

References:

Leffingwell, Dean, 2007, *Scaling Software Agility: Best Practices for Large Enterprises, Kindle edition,* Addison-Wesley Professional.

Schwaber, Ken, 2007, *The Enterprise and Scrum,* Microsoft Press

Cohn, Mike, 2009, *Succeeding with Agile: Software Development Using Scrum,* Addison Wesley Professional.

Cohn, Mike, http://www.mountaingoatsoftware.com/blog/synchronize-rather-than-overlap-sprints (accessed July 23, 2012).

Leffingwell, Dean, Scaled Agile Framework Big Picture, http://scaledAgileframework.com/ (accessed July 24, 2012).

Patterson, Kerry and others, 2007, *Influencer: The Power To Change Anything,* McGraw Hill.

Release Engineering

A guideline for successful software release

Jayashree Nagaraja and Vadiraj Thayur
Jayashree_Nagaraja@McAfee.com , Vadiraj_Thayur@McAfee.com

Abstract

Releasing software to the world is as important as developing the software. All the effort put into developing the software could go in vain if the right process is not followed for software release. This whitepaper is an effort to highlight some good practices that could be followed to ensure software in the form of tools, executable or products are delivered in an efficient manner to reduce the probability of mistakes to minimum. This whitepaper also covers releasing software in the SaaS (Software-as-a-Service, which is termed as Security-as-a-Service within McAfee) world and also touches upon other commonly used means of releasing software.

Release throttling is another aspect covered in the whitepaper. This is the process of controlling the availability of newly released software to the world so that any initial issues are found and fixed with minimal impact to the customer base. This aims at preventing issues from flowing to the greater customer base thus increasing customer satisfaction and also reducing the effort in fixing the issues out in the wild.

Biography

Vadiraj Thayur is a Sr.Technical QA Lead at McAfee, currently working in the McAfee India Center in Bangalore. He has been working for the past 8+ years in different QA roles on Enterprise as well as SaaS products. He has also owned Release Engineering for the McAfee SaaS product for a couple of years.

Vadiraj is a Bachelor of Engineering in Information Science from VTU, Karnataka, India. He also holds an M.S. in Quality Management from BITS Pilani, India.

Jayashree Nagaraja is a QA Engineer at McAfee, currently working in the McAfee India Center in Bangalore. She has an overall Software QA experience of 4+ years. She has been working for the past 2+ years in a QA Engineer role on McAfee SaaS product. She currently owns the Release Engineering for that product as well. Prior to McAfee, she has worked for 2+ years in Mindtree Ltd, Bangalore in a QA Engineer role.

Jayashree has a Bachelor's degree in Computer Science from Bangalore University in India.

1 Introduction

Release Engineering is a very critical and important part, like any other stage of the PLC (Product Life Cycle). The efforts put in by everyone, right from conceptualization to development to testing could go in vain with one mistake in releasing the builds. The criticality of this task doubles if the process involves release to the live servers (on-premise or OEM or SaaS or to manufacturing unit). For instance, in McAfee SaaS (Security-as-a-Service), any released build goes live within a couple of hours and could typically reach millions of users within a day; a case where a lot of caution is necessary while releasing. The most important thing to be ensured by the Release Engineer is “Release the right files, Release the files right”. In simple words, it means that the engineer needs to ascertain that the files being released are the correct ones.

Another key area of Release Engineering is to control the release process in order to provide the newer version/patch to different sets of customers one after the other. This method helps the engineering team to monitor the release process and the status (success or failure) of the release for different customers.

This write up is an effort to highlight the guidelines/best practices to be followed for a successful software release.

2 Need for a ‘Release Engineering’ process

Build release process is generally considered as a simple task where builds are copied from the build room to various live servers with the help of ‘file copy jobs/scripts’. Of course it is a build copy task, but not as simple as it is misunderstood. Release Engineering is a process in itself which has well defined stages before the build goes live and reach the customers. It consists of simple, yet very important steps which are to be followed religiously to ensure a less error prone release.

Any new software which goes live (On-premise/Saas) or released to manufacturing unit or to OEM partner could get into two kinds of issues. One being errors/faults in the new software and the other being wrong files and configuration reaching the customers. Both the issues can cause enough damage to the customers in terms of data loss, unstable environment, loss of productivity/time/money etc, which in turn causes loss of business to the company which is releasing the software, damage its reputation, affect customer satisfaction and result in competitor advantage. The release process which we follow should be capable of preventing such situations in the first place and cater quick solutions to both kinds of problems in order to avoid the spread of damage. The simplest solution to both the problems is to rollback the changes and switch to the stable version of the software. The process we follow should ensure least turnaround time in such critical situations. However, in case of release to manufacturing unit or to OEM partner, if a post production issue occurs, it is not feasible to switch back to the older version of the product, as the release reaches the customers through the hardware, in the form of CD/DVD kit or pre-installed on the PC along with the Operating System. In such situations where the spread of issues cannot be controlled, preventing the issues by following a systematic release process would be a better choice.

There should be a lot of thought process involved in designing a release process which is methodical to prevent errors, which foresees the occurrences of post production issues, which has the ability to incrementally provide the new software/version to customers and has a backup plan to halt the availability of erroneous files without causing much damage. Hence, any software release calls for a better Release Management System and it is worth the effort!

3 Release Engineering Process

The Release Engineering process can be split into 3 stages: Pre-release process, Release process and the Post release process.

3.1 Pre-Release Process

3.1.1 **Team Approval**: Once the build has passed QA, it is necessary to get a "GO" for the release from all the members of the team including the Engineers, Managers, Product managers, Program managers and all others concerned. It is also a process of recording the confidence level on different aspects of the product release from concerned members of the team. For instance,

- Engineers (Dev and QA) express the confidence level on the features and functionalities of the new product,
- Managers acknowledge that all the committed features are implemented and certified,
- Product Managers certify that the new product caters the intended solution to the requirements/problems of the customers (based on the engineering confidence),
- Program Managers, who tracks the release timelines, approve the release dates.

There has to be a logical way of recording confidence level from the Dev and QA, as it forms the basis of consent from other members in the team. Easiest method is to ask each Dev and QA in the team to rate different features of the product on a scale of 10, based on the development, test and regression effort put in so far; defects logged, fixed and deferred; performance of the feature etc. The individual rating can then be aggregated to come up with the Feature Score for each of the features, which can again be averaged to arrive at the Product Score, which speaks the confidence level of the team. As the ratings of different features are aggregated, a poor rating for one of the features could be subsided by a good rating for the others. So it is important to set a standard of acceptance or a cut off number for individual feature rating as well as the Feature Score and the Product Score, below which an alarm has to be raised or a strong reasoning has to be provided to accept the dip in rating.

Pictorial Representation of Recording Confidence Level of Engineering Team

Product Features	Dev 1	Dev 2	Dev 3	QA 1	QA 2	QA 3	Feature Score (Out of 10)
Install	8	9	8.5	8	7.5	8	8.16
Antivirus	9	8	7.5	7	9.5	8	8.16
Firewall	7.5	8.5	9	7.5	8	9	8.25
Site Advisor	5	5.5	5.5	6	4	5	5.16
DAT and Product Updates	8	7.5	9	8	9	8	8.25
Upgrade to Newer Version	9	8	8.5	9.5	9	9	8.83
Product Score							7.80

Standards of Acceptance	
Feature Score should be	> 6.0
Product Score should be	> 7.5

This is just an illustration. Numbers and features are just indicative

It is important to note that, if there is a “NO GO” from any of the team members for any reason, it is necessary to find out the criticality of the issue being stated in order to decide if the product should be released now or postponed. Hence any release should be approved by the entire team before it reaches the customers.

3.1.2 **Informing the stakeholders:** Once the build has passed QA, all the stakeholders (both internal and external) including the Marketing Team, Sales Team, Support Team, Manufacturing Unit or OEM Partner, Infrastructure/Data Center Team, Partners and Customers should be intimated on the new product version and the details of the release. It is a necessary step to initiate the action items related to the release, on the part of the stakeholders (if any). For instance,

- The Marketing and Sales Team should be aware of the new features and functionalities of the product to market/sell the right solutions to the customers,
- The Support Team should also be trained on the product features to guide the customers on product usage. It is also important to publish the Product Guide, List of Known Issues, Troubleshooting Tips and the Supportability Document to effectively solve the customer issues,
- The Manufacturing Unit or the OEM Partner should be intimated for hardware readiness in order to reach the customers on time,
- The Infrastructure Team or the Data Center Team should be informed on the required configuration changes (Ex: Web Server setup) on the live servers to support the new product/version,
- Lastly, Partners and Customers should be intimated on the release, especially in cases where administrator/user interaction is expected for successful install or upgrade.

3.2 Release Process

The release process discussed here consists of various ‘Best Practices’ which can be adopted for any kind of software release. It is important to ensure that these steps are followed sequentially and methodically (as applicable).

3.2.1 **AV Scanning:** One very essential process that needs to be performed as part of the release process is to scan the release candidate build to ensure that none of the files being released get detected as malicious by any prominent Antivirus software in the market. To accomplish this task, two or three ‘Release Servers’ which are used for the release process should be installed with at least two different Antivirus software with latest signature updates and the build should be scanned to ensure that there are no infections. This step is of high importance for any software product release, as it ensures that the build does not contain any malicious files that affect the customers and that there are no malware patterns in the build. If any of the files in the build are modified or cleaned or deleted by the Antivirus software, install/upgrade/any other functionality of the product may not function as expected when it reaches the end user.

3.2.2 **Build Archiving:** Good archiving is another secret to successful release. It is necessary that the current files and configuration on the live servers are backed up on the Release Servers before starting the release process. All previously released versions of the product (at least the

versions which are still being supported) should be archived and stored in a build repository. Release logs and hash/comparison results can also be archived. These will prove very useful in situations requiring a rollback and also to trace back post release issues. They are also useful in situations when a patch has to be released on a supported older version of the software, as it provides a quick reference to the base build on which the files can be patched and tested on the QA environment before releasing. Hence it saves a lot of time and confusion while dealing with older releases.

3.2.3 **Release to Staging Environment:** Releasing the product to Beta servers to get feedback on the new product/version is a common practice in any software release process. However, the environment and the configuration might be entirely different on beta and live servers, which may lead to undiscovered configuration issues on production environment. To overcome this problem, it is a good practice to stage the final build on an environment which is equivalent to live servers and validate through a couple of basic tests to gain confidence on the files and configuration which is set to go live. The risk of post production issues due to configuration errors can be largely reduced with this step.

3.2.4 **Release to Live Servers:** A software release can be of various types including release to an On-Premise setup, to SaaS environment (Cloud), Release to Manufacturing Unit and Release to OEM Partner. For any kind of release, there has to be a systematic way of execution of this step which is designed with utmost caution. Considering that all the above mentioned steps are followed before reaching this stage, a generalized method which can be adopted by all different kinds of release is discussed here.

- **Release Sandbox:** Make a copy of only the build (or files) which is being released as part of the current release on the Release Server.

- **Release Candidate:** Make a copy of current live build (from the archived build repository), patch the new files onto it and make the required configuration changes (if any) to get the Release Candidate version ready.

- **Build Comparison:** Compare the current live build (as in the archived build repository) with the Release Candidate build to check for any major discrepancies. The difference set should be exactly same as the files in the release sandbox.

- **Copy files to live servers:** The build which is ready to be released should be copied from the Release Servers on to the Live Servers with the help of file copy scripts. The scripts that are used for copying the files are very critical. They should be tested a couple of times against dummy builds and servers to gain confidence that they achieve the correct results. As the files are copied from server to server during the release process, the copy needs to be verified at each and every step. This can be done by file/build comparison. One very common way of comparing binaries is by hashing. The files in both the locations are hashed separately and the hashes are compared. Typically, the hashes should match exactly otherwise, strong reasoning would be required to explain why the difference in hashes is acceptable. Many hashing tools are available which could be used. Alternately, some languages like PERL also provide hash modules. Sometimes, when a couple of files are being released to production, to achieve a quick turnaround, file comparison tools like Beyond Compare can be used as well. They can be used to perform file/folder level and binary comparisons to ensure the correctness of files and configuration.

3.2.5 **Post Production Validation:** Release process is not complete until the release has been verified with a couple of basic validations from the live servers. This step may simply involve a

download of a file, a basic installation of the product or a simple upgrade from previous version to the newer version, to ensure that the release has achieved the intended results.

3.3 Post Release Process

The post release process is the last and equally important stage of software release. It consists of two major steps, one being Phased Rollout of the software and the other being Post Production Validation.

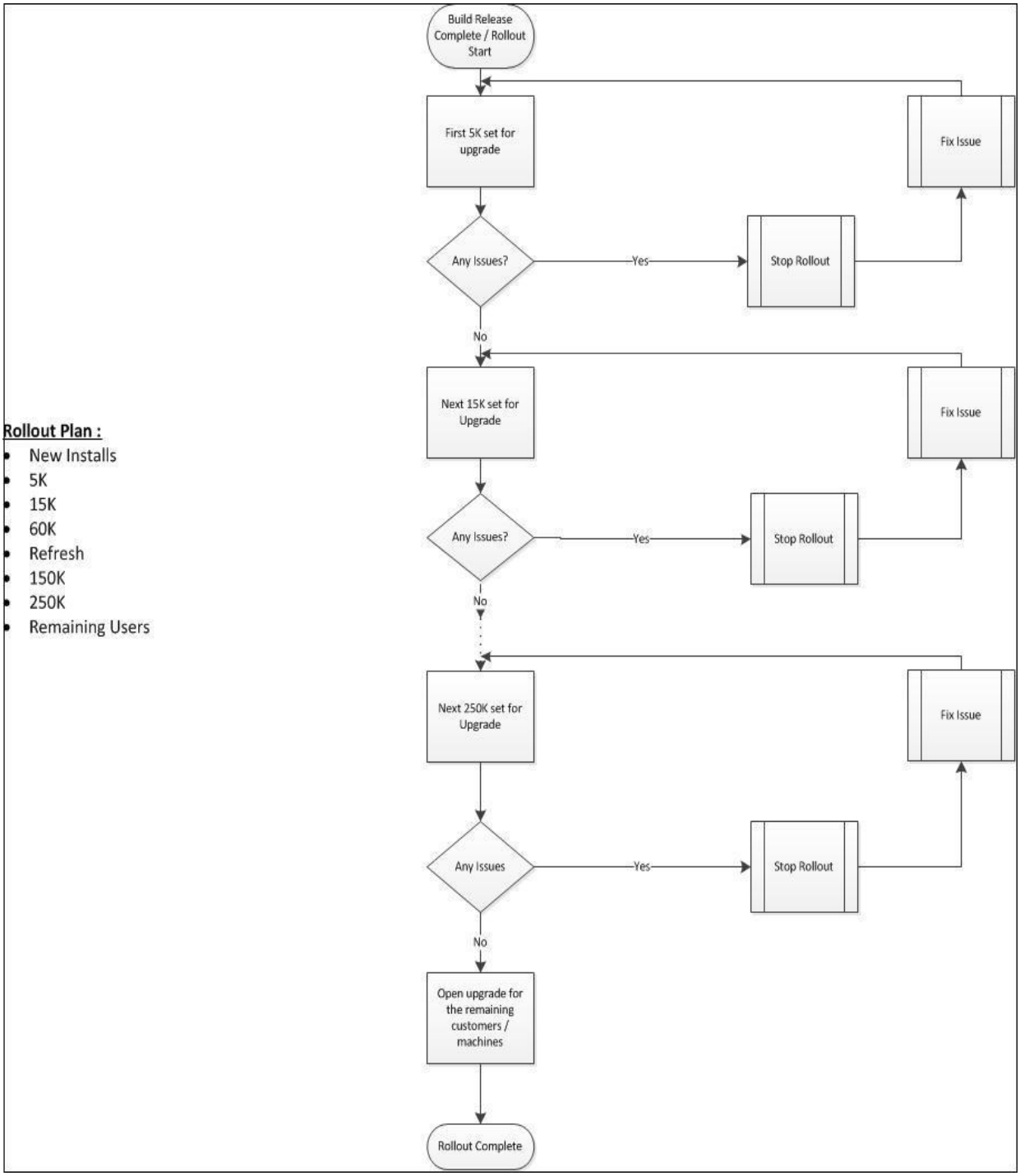
3.3.1 Phased Rollout: Once the build is successfully released, making the software available to all customers at once might be risky. If in case there are issues with the new release, all the customers will end up running into the same problem at once, which might require a bigger effort to deliver rectification. It is important to control and manage the availability of the new product version to the customers. The advisable method is to incrementally upgrade smaller sets of customers by rolling out the product in a phased manner. The importance of Phase Rollout is felt mostly by the 'SaaS based product teams', as any change which is released on the cloud would be available to all the customers immediately unless controlled. The other forms of release (like CD, OEM, Uploading onto a website etc.) would also benefit from the phased rollout process. The Phase Rollout mechanism is also helpful in cases where a specific hot fix has to be provided to only a few set of customers in a scenario where thousands of other unaffected customers are also pointing to the same live server on the production setup. The Phase Rollout of software is a very handy process which could be explained as below:

- **Identify the phases:** The customer base should be broken up into smaller segments to which the product should be released in a sequential manner. The breakup can be done based on various criteria like: Language, node / license count, complexity of their environment, products they use, Location etc. For instance, phased rollout for a software can be done based on languages, the English customers could be given the software first, then to German and French and then to Japanese and Chinese and so on. If the criterion is just based on numbers, the customers can be chosen for upgrade as, 5k customers at once, then 25k customers and then 100k customers etc.

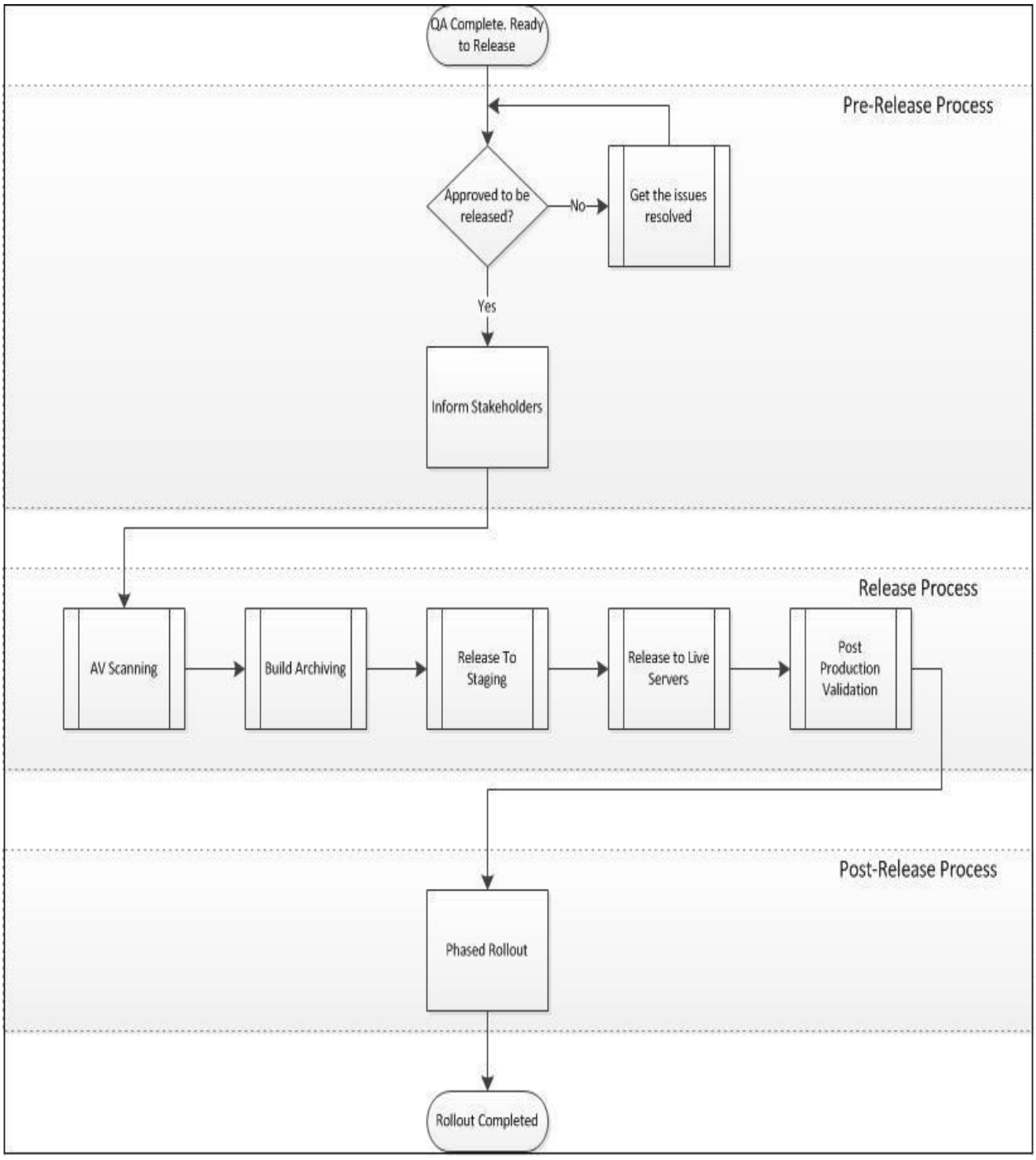
- **Rollout stage-by-stage:** Once the different phases are identified, the software should be rolled out in that manner. It is good to keep customers informed if they are being included in any phase of rollout so that they get prepared to move to the new version. In order to incrementally upgrade the customers, a backend (database) which is designed to support the rollout is a good thing to have. In this way, the customers to be included in phases can be selected in the database. Before beginning the phased rollout, it **i**s a good practice to test it in the QA environment to validate the correctness.

- **Review and continue the release:** After the first segment of customers receive the product updates, review the status of the release in terms of feedback from the customers. If there are any issues reported, they need to be fixed and verified on the customer environment before proceeding with the next phase. If there are no issues observed, the next set of customers should be upgraded and so on. With couple of such cycles, if there is enough confidence gain on the new product, software can be opened up for upgrade/download to all the remaining customers.

Pictorial Representation of Phased Rollout Process



4 Work Flow Diagram of Release Engineering Process



5 Role of a Release Engineer

The secret of Release Engineering is to 'Release correct files and configuration onto correct servers'. This is possible only if the release engineer is always up to date on the changes in the product, current requirements/specifications/configurations for the release and the production setup. It is also the responsibility of the release engineer to co-ordinate between the teams which owns the different aspects of the product, to ensure that all the related components are available on time and the integration functions as expected. For instance, to adopt Phased Rollout for a release, the DBA should be informed well before and the scripts required to split the customer base and rollout the product should be ready. Release Engineer play a vital role in discovering issues related to files and configuration on the production setup. Hence, a release engineer should be keen on finding the right answers for any small discrepancy observed in any stage of the release process.

In all, a good Release Engineer requires the following:

- Good knowledge of the product,
- Good knowledge of the production servers,
- Good knowledge of the release process,
- Hands-on experience on the tools used,
- Lot of emphasis to minor details,
- Patience,
- A positive questioning attitude.

6 Conclusion

Any software release demands a well-thought Release Engineering Process, to prevent the problems rather than finding a solution after the damage is caused. Here are a few advantages of following a good Release Engineering Process:

- Copying files on to intermediate locations on the Release Servers helps the Release Engineer validate the correctness of files at every step (less error prone),
- Release to staging environment provides an opportunity for pre-production validation,
- Rollback operation is handy with file and configuration backup,
- File archiving helps quick future references,
- Phased rollout helps early discovery of issues and prevents damage on larger customer base,
- Along with all these processes, a good release engineer is required to adhere to the processes and improve them over time.

If not planned and managed correctly, release process becomes very lengthy, messy, time consuming and risky.

For any kind of software release, the essence of Release Engineering remains the same:
"Deliver the correct files at the correct time in the correct way to the correct location"!!!

7 References

None

Speeding up Cross-Platform Testing – A Case Study

Jeffrey Weston
Jeffrey.Weston@microsoft.com

Abstract

Microsoft Office for Mac has a very efficient developer process for porting code across platforms (from Windows to Mac). Unfortunately for testers, it creates many new bugs across our entire suite of applications, requiring a large amount of resources and time to test and find them all. Because the bugs appear at random each time we port code, test cost reductions are incredibly difficult.

To overcome this problem, we investigated a sample size of bugs to find the root cause of the defect. We identified the exact code change that introduced the defect, collected data on the process the developer was following and on the code context in which the bug existed. Categorizing the bugs by common attributes produced bug patterns that we could then act upon.

This technique provided valuable information that allowed us to extinguish (or at least severely limit) an entire category of defects, scope our testing to only the riskiest areas, and identify which test tools and techniques would give us the most value in terms of finding the bugs after future merges.

This paper describes an overview of the project as well as a list of requirements, pre-conditions and a template that other teams can use to reproduce this root cause analysis method, enabling teams to identify their own bug patterns and make investments to help find the bugs more efficiently.

Using this root cause analysis method is an effective means to help test organizations reduce costs while increasing engineering and product quality.

Biography

Jeffrey Weston is a Senior Software Tester with over nine years of testing experience. He currently works in the Office for Mac group at Microsoft, where he is architecting a new test strategy to testing ported code. Previously he has architected and driven performance testing for Microsoft Office for Mac 2011, as well as tested and shipped three versions of Microsoft Word for Mac. His interests include software metrics, root cause analysis, and improving engineering quality.

Jeffrey Weston has a B.Sc. in Computer Science from the University of Victoria in British Columbia, Canada.

Copyright Jeffrey Weston 2012

1 Introduction

A Microsoft Office product like Word or Excel has existed on the Macintosh platform since the beginning. Up until 1997, Office for Windows and Office for Mac were co-developed at the same time by the same developers. In 1997, the Macintosh Business Unit was formed and a separate group of Mac-specific engineers took over.

In order to stay compatible with its Office for Windows counterpart, Office for Mac ports code written by Office for Windows engineers and integrates that new code into the existing Office for Mac code base. This enables new features and maintains compatibility across the suites for our customers. Internally, this porting and updating process is called “merging”.

The merging process is a very efficient means of getting code changes from Office for Windows into the Office for Mac codebase. The majority of the changes are automatically merged; the remaining work includes manually updating and integrating the platform-specific needs. Using this process, tens of thousands of code check-ins can be merged quickly.

However, just because a process is efficient for developers does not mean that it is efficient for testers. The bugs that result from this process can appear, from a black box perspective, completely at random. They can appear across the Office Suite while varying widely in severity.

Using a root cause analysis approach modeled on airplane crash investigations, I was able to understand why certain merge bugs occur, and come up with recommendations on how to prevent and find them more efficiently.

This paper will consist of the following:

- An overview of the merge process
- The test problems the merge process creates
- An overview of the root cause analysis approach used
- The bug patterns found from the analysis
- Recommendations on how to prevent or find those bugs more efficiently

2 General merging process

To understand where and how bugs come from, I thought it was vitally important to understand the process that can produce them.

The merging process works as follows:

1. Office for Mac applications have an established set of “core” source files that their developers always try to keep in sync with the Office for Windows codebase.
2. At a point in time, a “snapshot” of the Windows Office code is taken and the matching set of core files are merged into the set on the Macintosh side.
3. This merging produces two outcomes:
    a. Code changes that occur automatically
    b. Code changes that result in a conflict. (A conflict occurs when an Office for Windows developer has updated code in the same location as an Office for Mac developer. Conflicts must be resolved manually.)
4. At this point, the code often doesn’t compile due to missing or changed APIs, usage of the language that is unsupported by the Macintosh compiler, and missing code that was not included in the merge.
5. Developers then work to resolve conflicts, update integration points, and work though compiler errors. This process can take anywhere from days to weeks, and occasionally months.

6. Once the code compiles, developers run some basic build validation tests and fix any bugs those tests reveal.
7. Once the build passes the build verification tests, it is finally handed over to Testers.

3 “Random” bugs

Although the merge process is efficient for developers, it poses problems for testers. Developers can work on the code for weeks or months until a build is testable. Over that period of time, developers change hundreds of files, editing, deleting and adding thousands of lines of code. This code churn is not isolated to specific features, but occurs across the entire project.

Due to the breadth of churn, developers lose track of key risk areas or are unaware of high-risk changes, resulting in testers having to do a full black-box test pass in order to find defects.

Looking at churn statistics and summaries from the source control tool can also be problematic. There are two types of churn in the merge process:

1) Automatically changed
2) Manually changed code

Historically, code that is changed automatically to match the Windows code introduces fewer defects than code changed manually. Unfortunately, the source code change details do not differentiate between these two types of churn.

This large amount of churn makes bugs appear at random across the product rather than isolated on a single feature; for example, inserting a hyperlink could cause a crash, a memory could leak in a text styles component, or widespread yet subtle document corruption could be introduced.

Because of this perceived randomness of bugs, testers resort to running as many tests as possible across a wide breath of features. Blindly running all the tests isn’t an efficient process. An early internal study I did showed some teams spend over a day-and-a-half of testing for every one bug they find. When teams have exhausted their test cases, or simply run out of time, it can still be unclear whether they have covered all the key areas.

4 Software Crashes and Airplane Crashes

At the time, I was sure bugs did not get introduced randomly into the product. They are commonly introduced due to initial conditions, circumstances and actions. If I could understand how bugs got introduced into the product from the merge process, I could become better at predicting where these bugs will likely occur in the future and then provide a set of recommendations to prevent or find them quicker.

The technique I chose to deploy was root cause analysis, and I decided to model it based on how airplane crash investigators do their job.

Airplane crashes are not very predictable, either: they seemingly happen at random, but, like software bugs, they don’t. They occur due to specific events and circumstances, and it is the airplane crash investigator’s job to find out what those were. They have to determine how an extremely complicated system of parts and people failed; they have to understand what went so horribly wrong.

I found several parallels between finding the root cause of a plane crash and finding the root cause of a software bug that helped me through the process of understanding how merge bugs occur.

4.1 Not a single cause but a chain of events

An airplane is a complex system, just like a software product. When a fault in the system results in a problem, there are often multiple reasons for why that fault occurred. Take, for example, the crash of the Concord, also known as Air France Flight 4590. There were several probable causes of the disaster. (BEA 2002)

1) During takeoff, the tire of the Concord ran over a piece of debris from a DC-10.
2) The fuel tank became damaged from a blown piece of tire hitting it.
3) Broken wires in the landing gear ignited the leaking fuel.

A chain of events can also cause software bugs. For example, a bug where images failed to appear in a Word document had the following causes:

1) After the initial merge, the source file required many additional small changes.
2) During the tedious process of updating these changes, a piece of critical Macintosh code was deleted.
3) Execution of the code resulted in a failure when a document was opened with images.

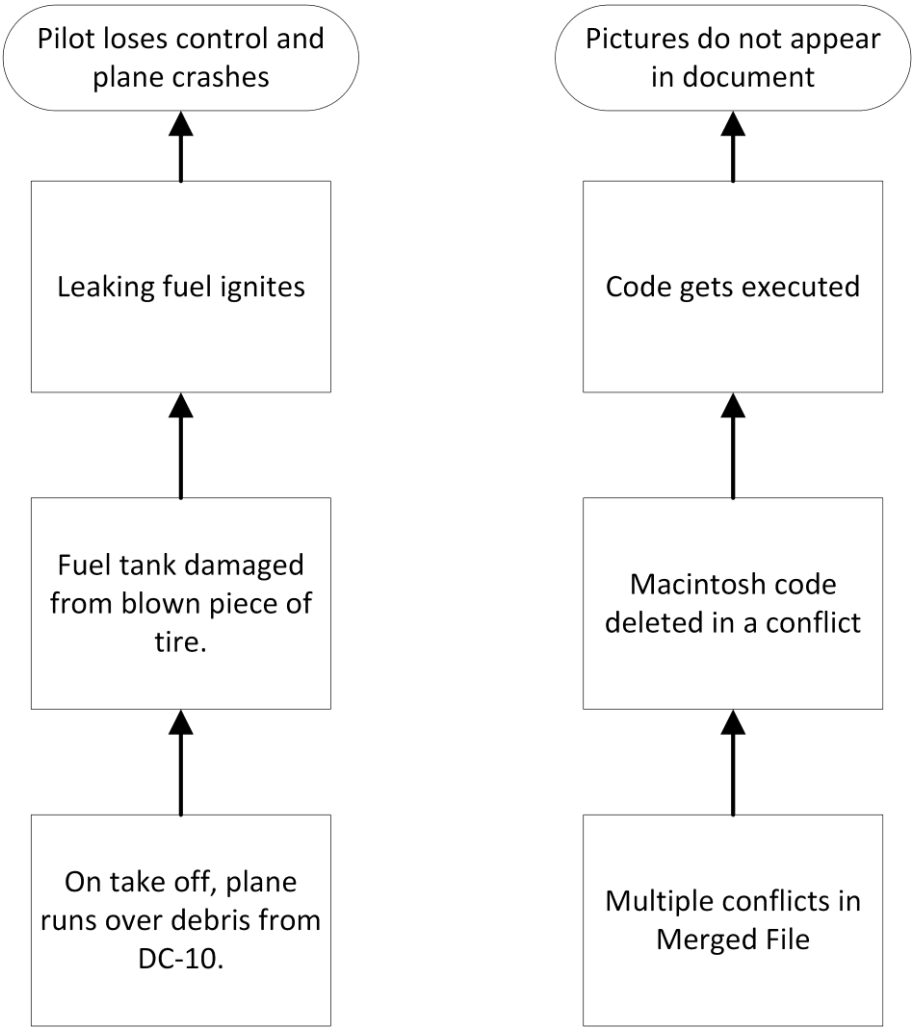


Figure 1 Chain of events leading to a plane crash and software bug

Understanding the full chain events allowed me to find solutions that could break the chain in multiple places instead of only preventing the initial cause.

4.2 Using logs to create a history of the event

Airplanes often have flight data recorders and cockpit voice recorders. The purpose of these tools is to help the investigators understand what was happening before and during the events in question. For example, the investigators of the Concord crash used the flight data records and cockpit voice records to know when the fire alarm first went off and how the pilots responded to it. (BEA 2002)

At Microsoft, we have a source control system that records everything we change in the code base.

Using the source control system, I started with the change that fixed the bug and the developer's comments associated with that change to go back through the history and find out when the bug was first introduced into the system.

The source control system could also identify the phase of the merge process in which the bug was first introduced by using the dates, the developer comments, and the type of changes being made.

1. Automatic merging of the code
2. Resolving a conflict
3. Fixing a compiler issue
4. Fixing a build verification test failure

4.3 Collecting information on the context of the bug

Flight data recorders and source control logs tell only one side of the story. Since my goal was to develop recommendations to improve the merge process, I needed to understand a lot more context and wider knowledge of the environment to make the most effective recommendations.

For example, crash investigators analyzed many parts of the Concord crash, all the way from the management structure of the Concord division of Air France and right on down to the position of the seats in the cockpit. (BEA 2002)

In software, I wanted to collect as much information as I could within the time constraints I had.

Some of the information I collected included:

The context of the code that the bug was in:

- Did the bug exist in code that was identical to Windows code?
- Did the bug exist in Macintosh code?
- Did the bug exist in code that was heavily mixed between shared Windows lines and Macintosh lines?

Tools that could have helped find the bug during the testing phase:

- If it was found manually, could it also have been found via automation?
- Could it have been found via code coverage in that area (i.e. by executing that code)?
- Could it only have been found via a human doing manual verification (i.e. a visual problem like pictures not showing up)?

4.4 Reconstruction

Sometimes, in order to understand what happened, investigators will reconstruct the plane from pieces of the wreckage. In the Concord disaster, they reconstructed the damaged wing to understand the cause of the fire. (BEA 2002)

In the software case, when I did not understand how a developer could make such a mistake, I reconstructed that part of the merge. This involved:

1. Learning from the developer how they personally performed the merge, such as the specific tools they used and the commands they executed.
2. Using the source control history to revert the source files back to the pre-merge state.
3. Executing that part of the merge myself.

The reconstruction process allowed me to see what the developer saw and have a better understanding of how the developer could have introduced the error. For example, I found that when some files are merged, they produce many conflicts that need to be manually resolved. However, the diffing tools some developers used did not provide all the information needed to resolve the conflict.

5 Findings

From the investigation, I identified four patterns of bugs introduced by the merge process.

1. **Incorrect conflict resolutions**: Manual conflict resolution often resulted in Macintosh code being hidden amongst matching Office for Windows code that was inadvertently removed.
2. **Churn when integrating Office for Windows changes into Macintosh code**: Our code has areas specifically marked as Macintosh code. Usually this code is within the same source file, or it is closely dependent on Office for Windows code. If the Office for Windows code changes, the corresponding Macintosh code must be manually updated to match.
3. **Churn when integrating Office for Windows changes into older legacy code:** Teams often have source files that were originally ported from Windows Office, but they are not merged regularly. These files become out-of-date and must be manually updated to conform to Windows Office changes.
4. **Ported Office for Windows bugs:** Not all the Office for Windows code we merge is fully complete. It can still contain bugs in incomplete features, and these bugs can simply be copied into Macintosh code.

The following sections explore the four patterns identified by the defect analysis process in order to explain how bugs in the merge process occur.

5.1 Incorrect conflict resolutions

A conflict can occur in a three-way merge. Word for Windows and Word for Mac start off from the same code (ORIGINAL file). WinWord makes a change in the code (WINDOWS DEV). Word for Mac makes a change in the same place (MAC DEV). The code is then merged together. A conflict requires the developer to choose which changes to accept or rewrite it to make it work correctly.

However, during conflict resolution, the developer may not see a critical line of Macintosh code. If the developer doesn't see any reason to keep the Macintosh code, they will often choose WINDOWS DEV to ensure the Office for Mac codebase matches Windows Office as much as possible.

This leads to Macintosh code getting deleted.

Example

```
        if (FPrinting() && list.Last < 0)
                EnumList(fFalse);

>>>> ORIGINAL ++++++++++++++++++++++++++++++++++++++++++++++++
        Assert(cache.isEmpty == fTrue);
        lserr = displayLine(theText, CLIPPED);;
        Assert(err == errNone);
==== WINDOWS DEV ++++++++++++++++++++++++++++++++++++++
        NewAssert(cache.isEmpty == fTrue);

        if (FInvalid(cache)
                err = errNone;
        else
                err = displayLine(theText, CLIPPED);
        NewAssert(err == errNone);
==== MAC DEV +++++++++++++++++++++++++++++++++++++++++++
        Assert(cache.isEmpty == fTrue);
        err = displayLineMac(theText, CLIPPED);
        Assert(err == errNone);
<<<< END +++++++++++++++++++++++++++++++++++++++++++++++
        destroyGroup(&theGroup);
        if (err != errNone)
```

In the above example code, a Windows Office developer has changed their assert API and added an additional `if` statement. The developer introduced a bug by accepting the WINDOWS DEV changes to resolve the conflict. This had the effect of changing the `displayLineMac(...)` API in the MAC DEV block back to `displayLine(...)`, which introduced a bug in Office for Mac.

Traits

These bugs tend to be:

- The result of simple human error (also known as pilot error).
- They are not specific to the feature being worked on.
- Increased risk of occurrence as the number of conflicts in the file increases.

Quite often the correct resolution is to accept the WINDOWS DEV block. If developers have to do this over and over, they can get fatigued and start missing some cases of Macintosh code hiding within Office for Windows code changes. This particular file had over sixty conflicts.

5.2 Churn in Macintosh code causes simple mistakes

Changes in Office for Windows code may require us to manually modify other parts of our code base. For example, Office for Windows modified an API called `DrawMark` to add another parameter. This function was also called in a Macintosh code block. After the merge, the code wouldn't compile because the function signatures no longer matched.

```
        #ifdef MAC_ONLY_CODE_BLOCK

        ...
        if (fDisplay)
             {
        <       DrawMark(pFoo, pBar, pMark, pLimit, pStart, pParam, pClip, pObj);
        >       DrawMark(pFoo, pBar, pMark, pLimit, pStart, pNew, pParam, pClip, pObj);
             }
             else
             {
                 ValidateEndmark(hwwd, prcw, emk, dypAltSpacing);
             }

        #endif // MAC_ONLY_CODE_BLOCK
```

The developer had to manually update this function call to use the newly added parameter. However, the developer made a simple mistake and inserted the parameter into the wrong position; the `pNew parameter` should be the second from last parameter, rather than the third from the last. Because the parameter types all still matched the signature, the code compiled and the developer assumed the change was correct.

5.3 Churn when integrating Office for Windows changes into older legacy code

Office for Mac contains source files that exist in Office for Windows and have not been freshly ported over in a very long time. Over the years, these files can diverge significantly from the Windows source and can no longer be merged automatically.

```
Original                          Changed

hbr = CreateBrush(pToolTip);      rtbr = NewCreateBrush(rtNil, pToolTip);
FillRect(hdc, prc, hbr);          NewFillRect(rt, prc, rtbr);
DeleteBrush(hbr);                 NewDeleteBrush(rt);
SetTextColor(hdc, pToolTip);      NewSetTextColor(rt, pToolTip);
```

In the above example, the `DeleteBrush` function was manually updated to use new API calls that the Office for Windows team implemented. This required manual changes that inserted a bug because the developer used the `rt object` in the `NewDeleteBrush(...)` function instead of `rtbr`. As a result, the program crashed executing the next line because the `rt` object was deleted by `NewDeleteBrush` invocation.

5.4 Porting over of Office for Windows bugs

When we merge code, we will undoubtedly port over bugs that have yet to be found or fixed in Office for Windows code. This pattern is the only pattern I discovered where there is no interaction between Windows and Macintosh code in Office. These bugs will most commonly occur when porting over large blocks of new code that are tightly coupled.

6 Recommendations to break the bug chain

Once I identified our defect insertion patterns, I developed recommendations for breaking the chain of events leading to discovering bugs late in the development cycle.

At each stage in the process, I tried to determine how to prevent the bugs being generated and what our teams could do to find them more efficiently—except, of course in the test phase, where the bug is already checked in to the code and thus cannot be prevented anymore.

The table below shows an example of the conflicts pattern.

Table 1 Bug prevention and targeting

Stage	How to *prevent* bugs from getting checked-in?	How to *find* bugs more efficiently?
Design Phase (Pre-Merge)	Prepare the code to reduce the number of conflicts before the merge starts.	Mark up unmarked Mac code so developers can clearly see the Mac code.
Implementing (During Merge)	Review each conflict resolution specifically looking for dropped Mac code.	Have Test Code Review the conflicts after check-in, looking for dropped Mac code.
Testing Phase (Post Merge)	N/A	Use code coverage analysis to determine whether conflicts resolutions were covered.

6.1 Reduce the possibility of conflicting code

Resolving merge conflicts is a manual and largely tedious process, which can easily result in simple mistakes.

One possible way to avoid the tedious work is to prevent it from being required in the first place. If one knew—in advance of the merge—of conflicting changes, such as changing the assertion API from "`Assert(...)`" to "`NewAssert(...)`", one could manipulate the files before the merge to prevent conflicts from occurring.

The following process shows how to edit the ORIGINAL file to match the WINDOWS DEV and MAC DEV and thus avoid the tedious conflicts:

1. Find the baseline file (ORIGINAL).
2. Find the new Office for Windows file (WINDOWS DEV).
3. Find the current Office for Mac file (MAC DEV).
4. Bulk-edit the MAC DEV file, changing `Assert` to `NewAssert` to match the WINDOWS DEV file.
5. *Temporarily* bulk-edit the ORIGINAL, changing `Assert` to `NewAssert` so all three versions now match.
6. Merge.
7. Revert the change to the ORIGINAL.

This process has already been integrated into our toolset and has shown to prevent up to fifty percent of the conflicts.

6.2 Proactively markup the unmarked Macintosh code before the merge

Bugs occurred during conflict resolution because the Macintosh code was removed erroneously, often because it was embedded in Windows Office code that was updated and not identified as Macintosh code.

If we were to clearly mark the code as Macintosh code—by using compiler `#ifdef` flags or comments, for example—it could increase the likelihood a developer would see this code and preserve it when the conflict is removed.

Marking all Macintosh code, however, would be costly. It could take several hours per file to do this and there could be hundreds of files. For projects that have a lot of source files that get merged, this could become very expensive.

6.3 Concentrate code reviews on conflicts

Code reviews can be effective at preventing bugs. However, the volume of changes that occur during a merge is far too large to be reliably reviewed. This has resulted in only cursory reviews or selective reviewing of specific changes.

Reviewing only conflicting change resolutions could prevent defects repeatedly being inserted during conflict resolution.

Code reviews could be very effective here, particularly if reviewers were instructed to focus on looking for deleted Macintosh code.

If reviewers were watching for only one thing, they might be more likely to detect it.

6.4 Target Macintosh code in specialized test suites

Based on the bug context data collected, this study revealed that bugs tend to be introduced where developers must manually update the code after a merge, which is required whenever Macintosh code is intermixed with Windows Office code.

Having a more concrete theory on where bugs tend to be injected gives us a better idea of where to focus testing efforts. With this knowledge, testers could mark existing tests or create new tests to specifically target merge testing.

Code coverage can tell us whether or not tests exist for these higher risk areas of code. Testers could create sets of merge regression suites to target the higher-risk Macintosh code blocks. Developers could then run these larger more targeted sets of tests in addition to their build verification tests.

Developers running and finding their own bugs is more efficient than having a test engineers find, reproduce, and log a bug and then verify the fix afterwards, particularly if the bugs are simple basic functionality issues (as is the case with most merge bugs).

6.5 Use bug lists to identify bugs in unfinished Windows Office code ported to Macintosh

Bugs can exist in code we port over from Windows Office when it is incomplete.

It is not realistic to try and prevent these bugs since their creation is out of the Mac Office engineer's control.

To find them more quickly, a Mac Office tester could look through the bug database on the Windows Office side to see what bugs are still active at the time the code was protected.

7 Conclusion

Before the merge study, our team did not know how to optimize our testing or improve the quality of newly merged code, but we now have a clearer idea of how to tackle those issues.

Root cause analysis revealed that a chain of events could cause bugs. By leveraging our source control logs, collecting data on the context in which the bug occurred, and reconstructing the events that caused the bug, I was able to identify bug insertion patterns. I used these patterns to propose recommendations for both preventing bugs from being generated where possible, and to detect other bugs earlier in the merge process.

Based on these findings and recommendations, our development and test teams are now investing in:

- educating the teams about the patterns and recommendations,
- changing manual procedures to make conflicts more accessible for reviews,
- automated tools such as code coverage and scripts to detect deleted Macintosh code.

By treating software bugs in a similar way to airplane crashes, I was able to turn a seemingly hopeless situation (random bugs) into an actionable one for our team. The recommendations I developed provide a clear direction on where to invest our time and energy in order improve the quality of our code base.

References

BEA. *Accident on 25 July 2000 at La Patte d'Oie in Gonesse (95) to the Concorde registered F-BTSC operated by Air France.* Investigation, Bureau d'Enquêtes et d'Analyses pour la Sécurité de l'Aviation Civile, Paris: Bureau Enquêtes-Accidents, 2002.

An Enterprise Framework for Evaluating and Improving Software Quality

Philip Lew
philip.lew@xbosoft.com

Abstract

With the world's economy increasingly driven by software products, there has been a relentless pursuit of software quality with research in model development, agile methodologies, and quality measurement. However, attention to and attempting to improve the quality of the product or its development process alone may not be the most effective means to improve quality. There are many other parts of the enterprise and various other factors that influence quality, especially where user perception of quality (sometimes called "quality in use") may be somewhat different than the product quality in a laboratory environment. Although requirements are often a cited reason for product quality problems, the quality of the sales process can also have a direct influence on requirements (i.e., custom software), and thereby influence quality. Customer service can also have great influence on the customer perception of the product's quality. Considering the many standards for software product quality, quality frameworks, and development models, we propose a framework called the Enterprise Quality in Use as an alternative view —including other parts of the organization's processes and outputs to improve quality not only in development phase, but also through each phase of the product lifecycle.

Key words: Quality model, Quality in use, Enterprise quality model, Software quality improvement.

Biography

After working for over twenty years in various management and technical positions in product development and management, CEO Philip Lew now leads the direction and strategy of XBOSoft (www.xbosoft.com). Phil works with organizations to assess the quality of their software, examine software quality processes, and set forth measurement plans so they can consistently improve software quality using systematic methods. He has authored articles in IEEE and ACM journal publications and trade journals; presented at several conferences on software usability, user experience, and quality evaluation; and is conducting post-doctorate research focused on software quality measurement. Phil has a B.S. and Master of Engineering in Operations Research from Cornell University and Ph.D. in Computer Science Engineering from Beihang University.

1 Introduction

As software becomes more ubiquitous in our everyday lives, shorter development cycles put pressure on software product quality. Abundant research and standards development has occurred in the areas of software quality, software quality models, and software quality processes. The ISO 9000 family of standards was developed to assist organizations implement quality management systems [2]. Similarly, ISO 25010 [3] was developed for use in the field of software engineering — to help organizations identify relevant quality characteristics for establishing requirements, their criteria for satisfaction and the corresponding measures specifically addressing product quality and quality in use (QinU).

Despite research focused on modeling and improving and organizations' capability to build quality products (ISO 9000 and TQM [1]), and standards addressing the evaluation of software products' quality (ISO 25010), a gap exists in the area of measuring and evaluating the end user's view on quality as influenced by the organization. Development models such as Agile, Scrum, Spiral, V-model, and others begin at requirements and end with acceptance testing. Yet there are other parts of an organization and its processes, prior to requirements and after acceptance testing, that can significantly influence a customer's perception of software quality.

To address this shortfall, this paper proposes a novel quality framework for examining and improving software quality — the Enterprise Quality in Use (EQinU) framework. EQinU is a flexible framework that can be used in any organization based on concepts similar to the ISO 25010 where the outputs of one phase of quality influences the quality at the next phase.

2 Related work on development and quality models

This section discusses quality and development models to provide a general background and lay the foundation for EQinU which is based on similar concepts.

2.1 ISO 25010 Quality Model

ISO 25010[3] is the newest standard on system and product quality models. You can think of a model as a way to break down abstract concepts such as quality into something we can get our hands around. The ISO 25010 [3] standard's views of quality can be summarized as follows:

1) Product Quality (PQ) - Specified by a quality model (i.e. a set of eight characteristics–Functional Suitability, Performance Efficiency, Compatibility, Usability, Reliability, Security, Maintainability and Portability - and a set of sub-characteristics per each characteristic are prescribed), as shown in Figure 1.

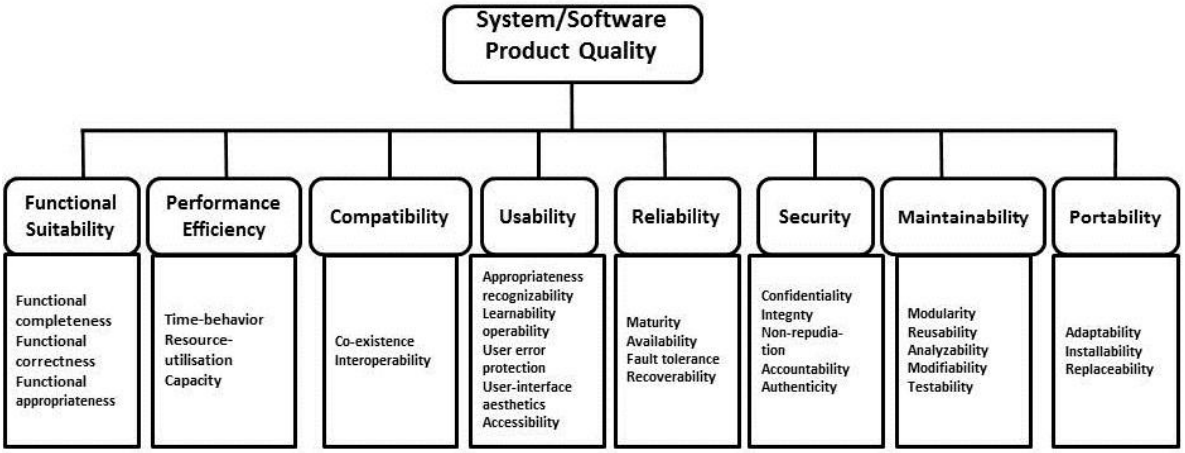


Figure 1. ISO 25010 product quality model [3]

The ISO product quality model defines quality characteristics such as Usability by using sub-characteristics, i.e. Learnability. This is a common method where decomposing a characteristic into sub-characteristics enables us to better understand the meaning of the characteristic.

2) Quality in Use (QinU) - Specified by a quality model (i.e. a set of five characteristics–Effectiveness, Efficiency, Satisfaction, Freedom from risk and Context coverage) as shown in Figure 2. Note that each characteristic can be measured and evaluated by the extent to which specific user needs in an actual, specific context of use are met.

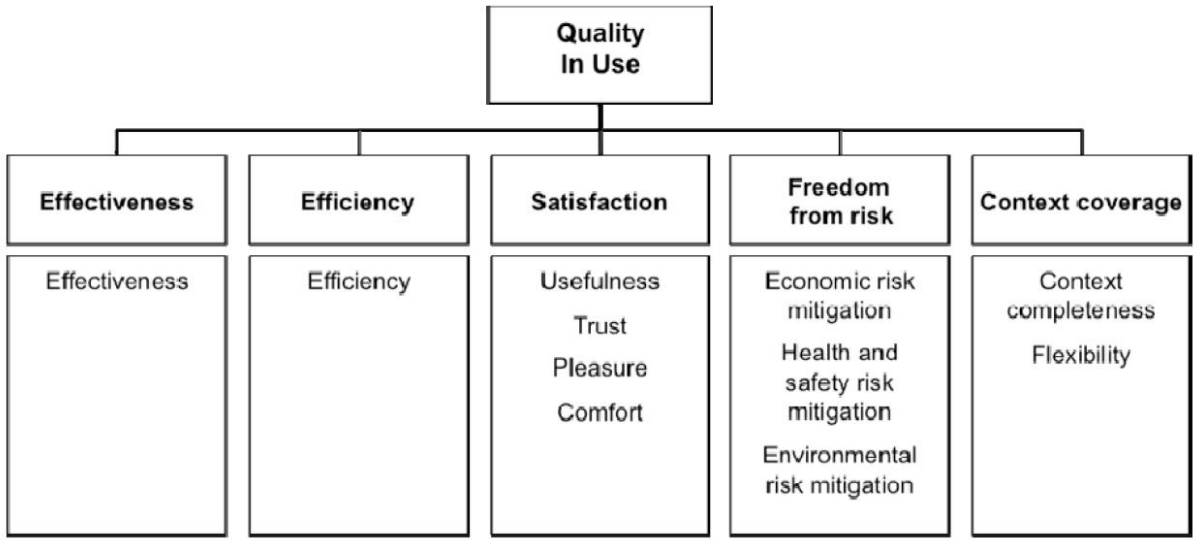


Figure 2. ISO 25010 Quality in Use model [3]

The ISO 25010 QinU model was developed to clearly differentiate product quality from the product's effect in a real situation of use. A product could have 'good' quality at a product level, with very good Performance Efficiency, yet 'in-use', Satisfaction could be very poor. As an example, suppose that a software application was designed to have the menus on the left rather than at the top. From a product quality point of view, it may fully satisfy user interface aesthetics criteria. This design may also satisfy Functional Completeness and Correctness criteria as well. However, from the user point of view, the unexpected menu location may negatively impact the Operability and Learnability of the software. The user perceives the application as having very low Efficiency.

Quality models can be used to specify and evaluate software quality from different perspectives in the acquisition, requirements definition, development and evaluation of software. In practice, depending on the domain and the end users, when modeling quality we typically include a handful of characteristics and sub-characteristics that are most important to the evaluator. For instance, a person in a purchasing department may use the model to specify requirements for each characteristic that vendors must adhere to. For example, under Performance Efficiency (time behavior), all printing response times shall be less than 3 seconds for the first page. For a person evaluating software usability, they may use only one characteristic of the model and add more precision and depth only to the usability characteristic. For a stakeholder involved in software design and allocation of resources, they may put different weights on the different characteristics of quality depending on the domain of their application. Rather than saying 'quality is important', quality models give us a means to better define our meaning.

My previous research utilized the Product Quality/Quality in Use paradigm and developed a flexible framework called 2Q2U (Quality, QinU, actual Usability and User experience) designed for evaluation of Quality in Use [4]. For 2Q2U, we used the ISO 25010 premise that Product Quality (PQ) influences QinU. For instance, if help is contextually based, then this influences the user's ability to learn the software when using it for a particular task. In this research, it was found that some characteristics, if improved at a

product level, definitely influenced performance at the user level. As shown in Figure 3 [5], if help completeness is improved, this will lead to improvement in the users' ability to complete tasks.

Related Quality in Use Attribute	Related External quality Related Attribute
Learnability in use: Sub-task completeness learnability	1.1.2.2 Learnability.Helpfulness.HelpCompleteness
	2.1.1 Information quality.InfoSuitability.Consistency
Effectiveness in use: Task Successfulness	1.2.1.2 Ease of use.Controllability.StabilityofMainControls
	1.1.2.2 Learnability.Feedback Suitability.Task Progress Feedback Appropriateness
Efficiency in use: Sub-task completeness efficiency	1.1.1.3 Learnability.Feedback Suitability.Entry Form Feedback Awareness
	1.1.2.1 Learnability.Helpfulness.Context-sensitve help availability
	1.2.3.1 Ease of use.Data Entry Ease.Defaults
	1.2.3.2 Ease of use.DataEntryEase.MandatoryEntry
	1.2.3.3 EaseofUse.DataEntryEase.ControlAppropriateness
	2.1.2.1 Information quality.InfoSuitability.InfoCoverage.Appropriateness
Effectiveness in use: Sub-task completeness	2.1.1 Information quality.InfoSuitability.Consistency
Effectiveness in use: Sub-task correctness	1.1.2.2 Learnability.Helpfulness.HelpCompleteness
	1.2.2.1 Ease of use.Error Mgmt.Error Prevention
	1.2.3.1 Ease of use.Data Entry Ease.Defaults
	1.2.3.3 EaseofUse.DataEntryEase.ControlAppropriateness
	2.1.2.2 Information quality.InfoSuitability.InfoCoverage.Completeness
	2.1.2.1 Information quality.InfoSuitability.InfoCoverage. Appropriateness

Figure 3. Relationships between Quality in Use (end user quality) and product quality attributes [5]

Using Product Quality and QinU models, Figure 4 shows the 'influences' and 'depends on' relationships from ISO 25010 [3] where one phase influences quality at the next phase including process quality as well.

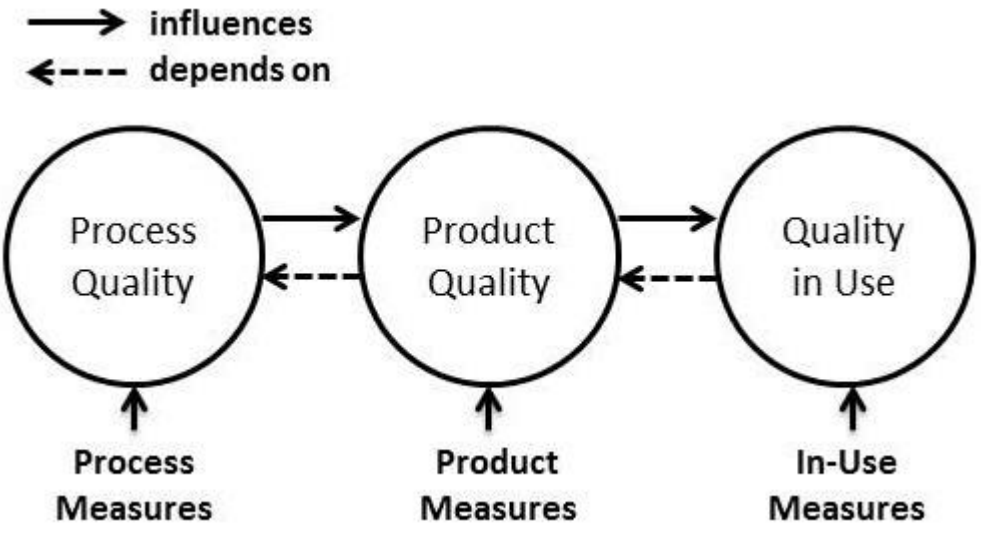


Figure 4. Quality in the lifecycle

The ISO 25010 standard and quality models are very general and most practitioners only use them as a guideline or starting point for what to consider when modeling and measuring quality. ISO 25010 model

usage states: “tailor the quality model giving the rationale for any changes.” What we can take away from this discussion on ISO 25010 are 3 main points:

- One phase of quality can influence the next phase as shown in Figure 4. We all know from CMMI that great processes do not necessarily lead to great product, but they do have influence.
- In practice, in previous research, we have been able to demonstrate a positive correlation made between product attributes and quality-in-use performance, thus showing the usefulness of this phase by phase ‘influence’ modeling concept.
- Modeling quality using a hierarchical tree format makes it easier for us to conceptualize and understand what quality is, which is the first step toward improvement.

2.2 ISO 9000 Quality Standard

- The ISO 9000 [2] family of standards was developed to assist organizations with implementing and operating effective quality management systems. ISO 9000 is founded upon eight quality management principles. The principles most applicable to our quality modeling work include:
- *Process approach*: Efficiently achieving desired results through activities and related resources that are managed as a process.
- *System approach to management*: Identifying, understanding and managing interrelated processes as a system which contributes to the organization's effectiveness and efficiency in achieving its objectives.
- *Continual improvement*: Continual improvement of the organization's overall performance should be a permanent objective of the organization.

The ISO 9000 standard has a broad and general reach and can apply to all organizations striving to increase the quality of their products and services by applying these principles in their operations. However, it does not contain details on quality characteristics or using decomposition as a means to modeling and evaluating quality. What we can take away from ISO 9000 is:

- Quality practices should be applied to the organization as a whole and not just one specific department.

There are other models similar to ISO 9000, including Total Quality Management [1]. These methodologies and models are oriented towards general quality processes and organizational capabilities to develop quality products and services, but still lack quality modeling characteristics and sub-characteristics for more specific understanding such as in ISO 25010.

2.3 Development models

There are many development models, but the objective and framework of the models are generally designed to solely model and measure software products or processes within the sphere of influence of the development organization. The V-model [7], for instance as shown in Figure 5, only shows requirements through acceptance testing.

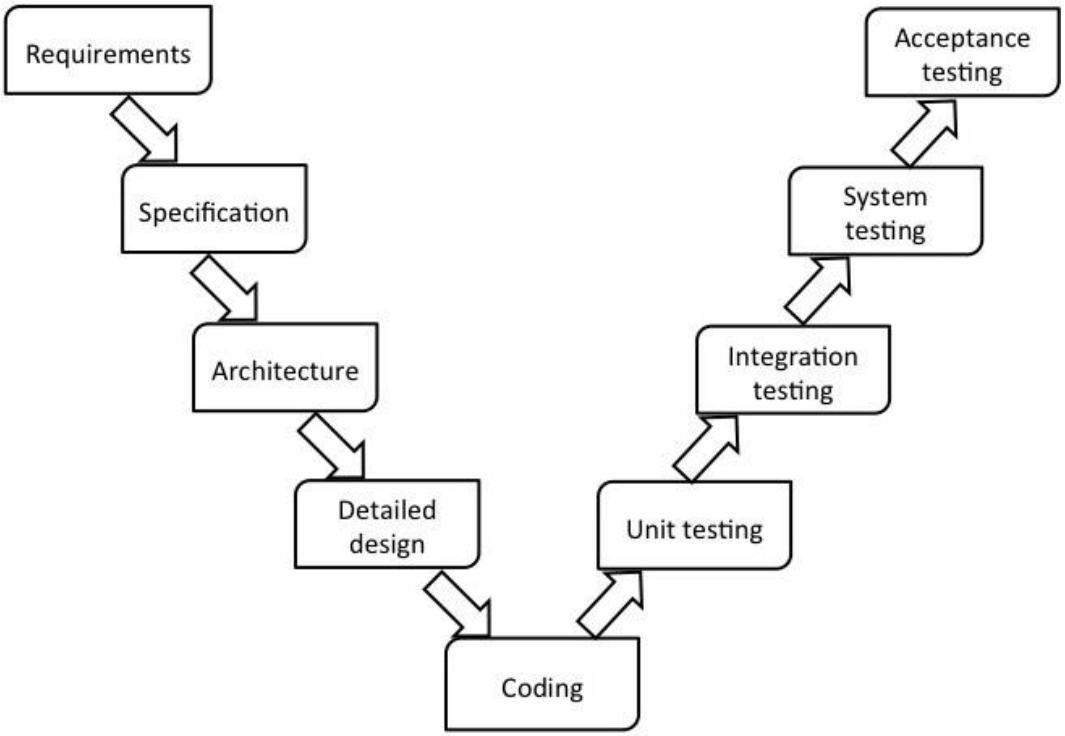


Figure 5. Simplified V-model Development Paradigm

The V-model and others including agile, waterfall, spiral, etc. all primarily focus on development and QA as the primary drivers of quality. A few critical elements are lacking:

- The end user's perception of quality is influenced by many other factors outside of development and quality assurance.
- Other parts of the organization also influence the quality of not only the product itself, but also the user's perception of quality.

3 EQinU Framework

In developing the EQinU framework, we kept in mind an overriding philosophy not to just set up a framework for a good product, but for a quality product from the end user's perspective. We also considered the main points of the previous discussion:

- Quality should be modeled in a hierarchical manner for ease of understanding and therefore improvement.
- Quality should incorporate the processes from other parts of the organization, not just development and quality assurance.
- Quality from one phase in a product's lifecycle development can influence the quality at the next phase.

Given this, let's examine a typical product lifecycle, as shown in Figure 6, where a company conceives of a product, then selling and producing it, and finally maintaining and servicing it. During the course of maintaining and servicing the product, the company runs into new opportunities for different or adapted products and the cycle begins again.

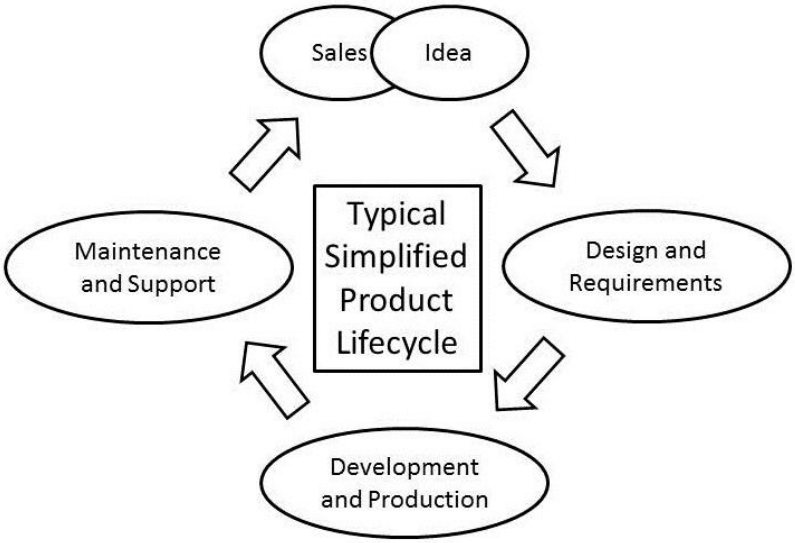


Figure 6. Typical Product Lifecycle

Now, we take the concept from ISO 25010 where one phase of quality has influence on another downstream phase, and each phase's quality is dependent on the output of one or more previous phases. In a general sense, quality of the outputs of phase N influences quality at phase N+1. By transforming Figure 6 from a cycle into a linear production line, this is conceptualized in Figure 7.

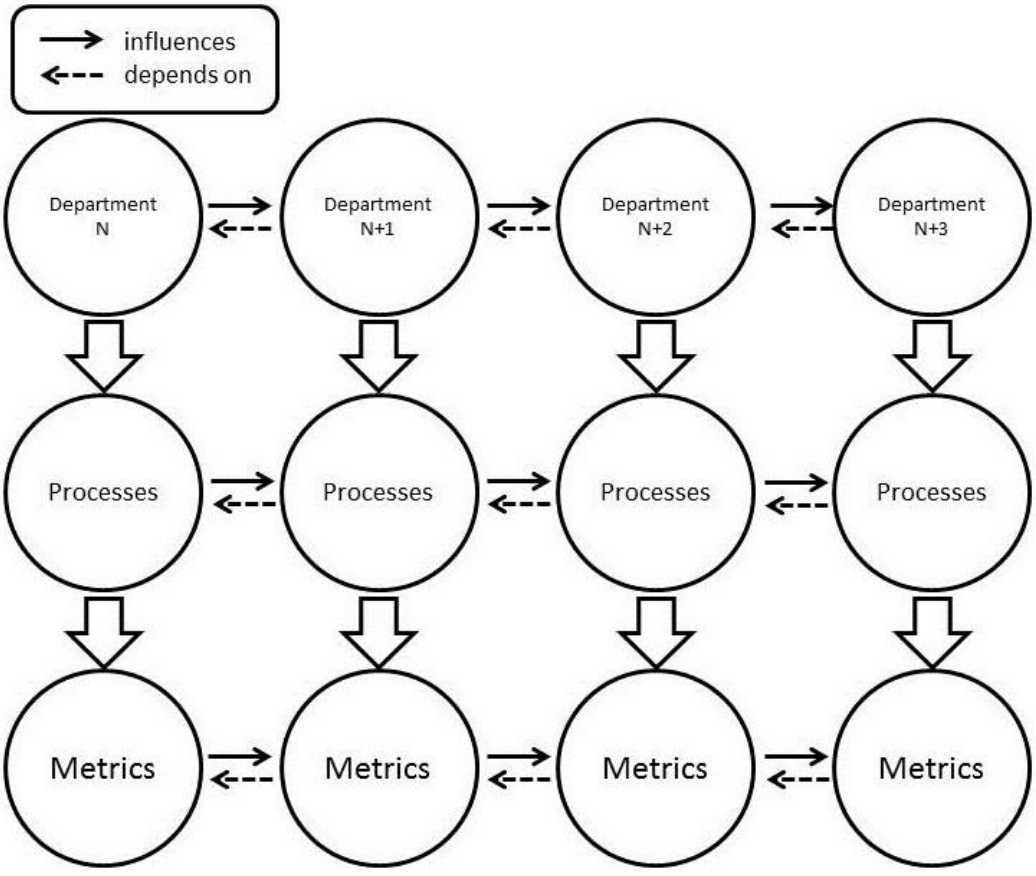


Figure 7. Proposed EQinU modeling framework.

More specifically, we instantiate the framework with a more specific model oriented towards organizations producing software as shown in Figure 8.

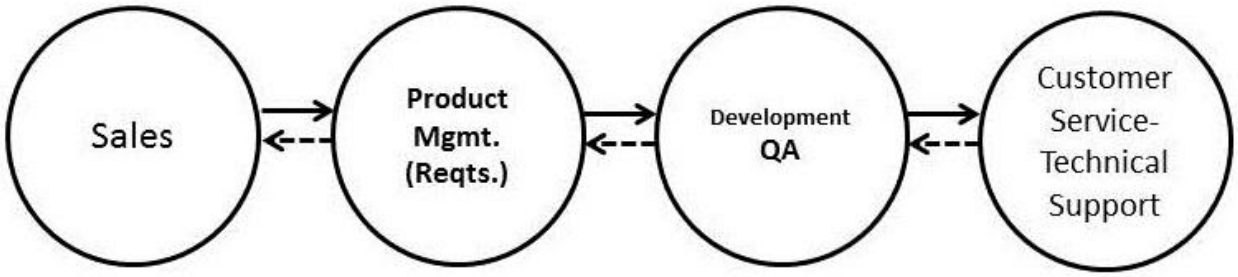


Figure 8. EQinU Software company general instantiation.

This extends the V-model forwards and backwards in an entire software development organization, where sales processes can influence the quality of product requirements, and in turn, the product requirements processes can influence the quality of development. Further in the product cycle, customer service or technical support can give negative impressions to an existing customer by entering a service ticket either incompletely or incorrectly such that it takes longer than it should to resolve the issue, or the customer needs to call again for the same issue. It's easy to see that the output of one process influences the performance and outputs of the next process down the line.

Therefore, the concept of a Total Quality Lifecycle of a product, from inception to usage by the end user, should model not only the product quality, but also phases prior to the product, and after the product has been developed, as shown in Figure 9.

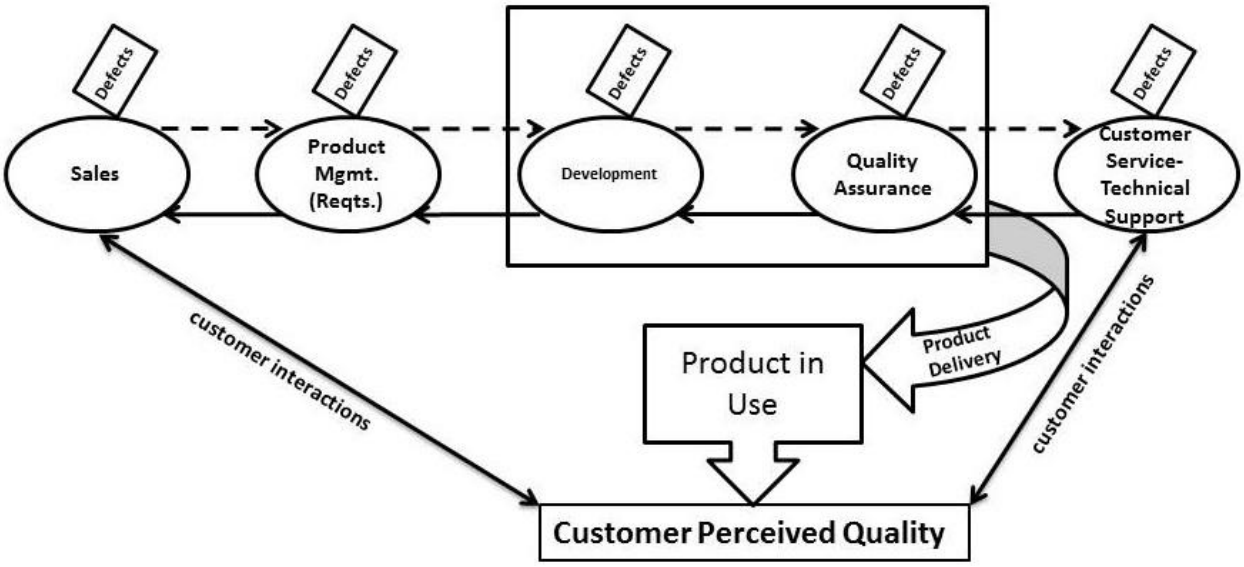


Figure 9 Total Quality Lifecycle

Each phase produces output, and that output could contain defects influencing later phases of the entire process that in the end, affect customer perceived quality. Below are a few examples:

- Sales: Sales presents the product and its features and capabilities to prospective customers. If they present the product such that after the sale, customer expectations are not met (for example

delivery date or product capabilities), then this could contribute to a low perception of product quality. Misrepresentation in its various shapes and forms could be considered a sales defect.

- Product Management (Requirements): Product management is responsible for many things and one of them is gathering from customers features to prioritize and include in the product roadmap. When features are specified and requirements are written and handed off to developers, if the developers do not understand the requirement fully, or must go back to ask the product analyst for clarity, this constitutes a requirements defect. Additionally requirements that are incomplete or not captured also represent a defect the same way as if you delivered a drawing of a table to a furniture maker that only had three legs.
- Development: These are familiar defects that already have plenty of metrics and measurements. A developer writes code according to a requirement or user story, and there is an error in the code causing behavior to be different from the requirement. Derivatives of these defects could be defect regressions or defects that, when fixed, create other defects.
- Quality Assurance: Testers also produce defects in their work by not clearly documenting defects so that developers can understand them or not thoroughly investigating an incident such that the defect cannot be reproduced. Note that this is a defect in testers' work product, not the product itself. Ultimately, errors or defects in the output from QA can affect the quality of the end product.
- Customer Service/Technical Support: Support and service representatives can create defects when they take calls from customers and give incorrect or incomplete information causing a customer either not to be able to solve their problem, or need to call back or both.

4 Discussion and usage of the framework

As experience with CMMI has shown, stringent processes and documentation do not guarantee a quality product. The EQinU framework is not intended to be a standard, or anything similar to CMMI where all the 'I's need dotting and 'T's crossed with an auditor by your side. Rather, it is intended for use at a departmental level, to uncover elements that can influence quality, in particular in departments that traditionally were thought to have minimal influence on quality.

4.1 Field example: sales and product management

You may think that a sales call and marketing literature are unrelated to customer satisfaction, but we found in working with clients that quality problems can start way before development. Let's look at an example as shown in Figure 10.

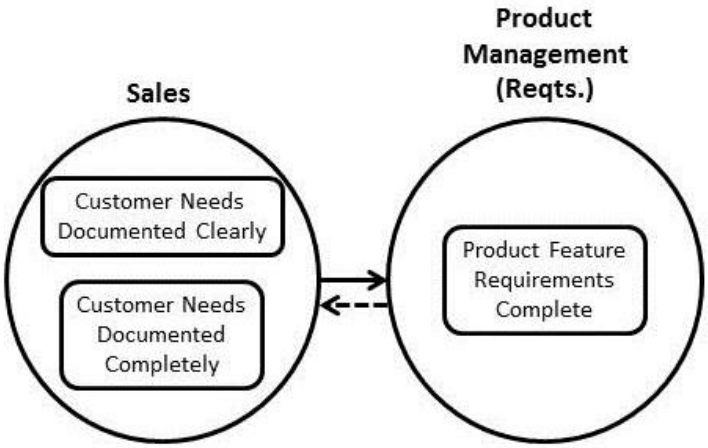


Figure 10. Sales Defects Influence Product Management (Reqts.)

Let's assume that the salesman, in a hurry to make the sale, makes a few mistakes:

1. He forgets that the customer has a special requirement regarding Euro currency, for online conversion each day automatically converting the rate according to a certain website (Customer needs not documented completely).
2. The customer tells him that they need to have the software delivered before the end of the summer. The salesman documents this as, "must deliver end of Q3" (Customer needs not documented clearly).

As a result of these omissions and inaccuracies, the Product Management department has errors in their requirements. They pass these requirements to development. Using an agile development methodology, this might be caught if the customer is deeply involved, but there is also a distinct probability that it may not be discovered until the product is delivered. Delivering the product on September 30, or even October 6, may be beyond what the customer thought of as 'end of summer' and there is no automatic conversion from Euros to USD. The customer is unsatisfied.

4.2 Field example: Customer service

If we look at defects in an organization, we can find them in many places other than those in development and QA. Let's examine an example in customer service where we were able to listen to sample calls from several customer service representatives and investigate the service tickets that resulted:

1. The customer called in because they had a problem posting changes from the client application into the cloud. After posting the changes, the system did not provide feedback that the changes were posted. They called into customer service to report the problem.
2. Because many people were having the same problem, there were heavy call loads that day. The heavy load caused many service representatives to take longer breaks, which contributed to both long wait times and exceptionally long call times. We noticed that sometimes, the representative would say "can you hold please while I investigate this error". But in fact, they were just resting, thereby extending call lengths.
3. When the customer service representative entered the service ticket, they forgot to ask what environment the customer was using (Windows, Apple, etc.) so the ticket was incomplete.

All of the situations above actually represent defects in other parts of the organization's work products that are not what we traditionally think of as software defects, but which can have a significant influence on the product quality and the end user's perception of quality

5 Conclusions and future work

In this paper, we have proposed a framework for modeling Enterprise Software Quality in Use (EQinU). In doing so, we have provided reasoning for extending the ISO 25010 quality modeling premise that phases in a product's development influence later phases and quality of later phases depends on earlier phases. EQinU extends this premise in the organization — into customer service and sales. As such, quality can be influenced, by areas of the organization other than software development and QA.

To illustrate the applicability of the proposed approach, examples from real clients in the field were presented to demonstrate the need to view software quality from a different mindset. It is no longer solely the job of development and quality assurance, but the entire organization, including sales and technical support to produce high quality software.

Ongoing research is focused on further utilizing the EQinU framework to model and understand the relationships among processes in an organization, their influence on product quality, and ultimately the end user's perception of software quality using measurement and metrics. In [6], we used survey methods to directly correlate quality metrics to end user satisfaction and we hope to extend that same

principle in this line of research. In the end, our goal is to discover that improvements or decline in performance in a characteristic of one organizational department influences the performance of another. Example correlations could include:

- Technical support call length and defects resolved.
- Time to fix a defect and number of calls to technical support.
- Customer needs documented accurately and completely by sales and product management's feature requirements.

The above are just examples of how one phase in the product lifecycle can have an influence on the end user's view of quality and that one weak point in an earlier phase can ultimately have impact on customer satisfaction and the end user's perception of quality. With this alternative paradigm in viewing quality, the amount of low hanging fruit may surprise you.

References

1. Deming W.E. et al, Total Quality Management, http://en.wikipedia.org/wiki/Total_quality_management.
2. ISO 9000, International Standard, Quality Managements Systems, 2000.
3. ISO/IEC 25010: Systems and software engineering. Systems and software Quality Requirements and Evaluation (SQuaRE). System and software quality models, 2011.
4. Lew P., Olsina L., Li Zhang, Quality, Quality in Use, Actual Usability and User Experience as Key Drivers for Web Application Evaluation, Lecture Notes in Computer Science, 2010, Volume 6189, Web Engineering, Pages 218-232
5. Lew, P. and Olsina, L., Instantiating Web Quality Models in a Purposeful Way, Lecture Notes in Computer Science, 2011, Volume 6757, Web Engineering, Pages 214-227.
6. Lew P., Qanber Abbasi M., Rafique I., Wang X., Olsina L.: Using Web Quality Models and Questionnaires for Web Applications Evaluation. IEEE proceedings of QUATIC, Lisbon, Portugal, 2012.
7. V-Model Software Development, http://en.wikipedia.org/wiki/V-Model_(software_development).

Lightweight Software Process Assessment and Improvement

Tom Feliz
Tom.Feliz@tektronix.com

Abstract

When most Software Engineers think of software process assessment and improvement, images of high paid consultants, byzantine process frameworks, and expensive audits come to mind. Established process frameworks, such as the Capability Maturity Model Integration (CMMI), are focused primarily on external certification and less on internal process improvement. Software process assessment and improvement need not be a heavyweight and burdensome endeavor. The best software organizations make process improvement an integral part of their organizational development and culture. The approach outlined in this paper describes a lightweight software process assessment and improvement approach that is practical, customizable, and can be implemented with reasonable effort. This approach is particularly relevant for smaller software organizations, which bear the costs of software process improvement (SPI) disproportionately.

For most software organizations, the value and benefit of assessing organizational maturity is in identifying tangible process improvement initiatives based upon well-established Software Engineering best practices. Considerable research has been conducted over the past 40 years into the practices that yield the biggest return on investment in regards to defect removal efficiency and developer productivity. Some of the most comprehensive research has been conducted by Capers Jones of Software Productivity Research (SPR). SPR has compiled a large database measuring the relative effectiveness of most common software practices. This data forms the basis of the software process assessment and improvement approach proposed in this paper.

Biography

Tom Feliz is a Senior Software Design Engineer and Software Quality Lead at Tektronix. He has been engineering software since he bought his first computer, a Commodore VIC-20, in 1983. Prior to joining Tektronix, Tom founded multiple technology startups and has worked in a variety of Software Engineering environments. His professional interests include Embedded Software Engineering, Hardware-Software Integration, and Software Quality.

Tom Feliz has a Master of Science in Computer Science and Engineering (MSCSE) from OHSU OGI School of Science and Engineering. He was awarded the IEEE Certified Software Development Professional (CSDP) credential in 2007, one of only a handful in Oregon.

Copyright Tom Feliz, October 2012

1. Introduction

Software process improvement (SPI) has enabled thousands of companies worldwide to produce higher quality software in a more predictable fashion while meeting cost estimates and schedule commitments [1]. SPI, based on well-known reference models such as CMMI [2] or ISO 15504 [3], provides industry best practice benchmarks that software organizations can use to assess their software process performance both for internal process improvement efforts and for evaluation by outside vendors. Historically, these efforts have been targeted towards large software organizations with hundreds of developers and thousands of employees [4, 5]. This has left many smaller software organizations questioning the role of SPI in their software development efforts. Small organizations, with less than 50 employees, account for approximately 90% of the software and data-processing companies in the U.S. [6] and small data-processing companies with five employees or less constitute 65% of the U.S. industry [7].

Even though many smaller software organizations see the need for SPI, their needs have been disproportionately ignored in SPI research literature [4, 7]. Large software organizations tend to have formal processes based on established process frameworks, as well as dedicated software quality staff to oversee the management of software processes company-wide. Smaller software organizations, in contrast, generally lack internal software process expertise and software processes are often ad hoc. For SPI to be successful in the smaller 90% of software organizations, a more lightweight approach is needed.

This paper begins with a basic introduction to SPI, software process assessment, and a discussion of where existing process frameworks fall short. A simplified and practical process assessment and improvement approach is then presented with real-world example assessment artifacts. The hope is that by demonstrating a practical, simplified, and customizable cookbook approach to SPI and process assessment, the author might begin to dispel the common myth that SPI is out of reach for the great majority of software organizations (i.e. the other 90%).

2. Software Process Improvement

2.1 Introduction to SPI

Even the most chaotic of organizations use some form of process, even if it's just pure code-and-fix software development [8]. Software processes range from total chaos, with each individual working in isolation in his or her own style performing any tasks in whatever order is deemed appropriate, to regimented processes, where each finely-grained action is recorded, measured, managed, and documented. For most organizations, the ideal process is somewhere is between these two extremes. Projects don't necessarily have to be planned to still be using processes [9], which may even be quite effective in some contexts. The so-called "peopleware" process approach is based on just this theory. The goal is to employ the most gifted and talented people and place them in a well-supported environment conducive to intellectual / creative work and a highly collaborative process will arise yielding high quality outputs [9,10]. Clearly, there is some validity to such an approach. But when repeatable long-term organizational performance is required with minimal risk, then a little process can go a long way.

There is a common belief, especially among smaller software organizations, that software assessments and SPI only burdens creativity and stifles innovation with unnecessary overhead. In this view, as in the "peopleware" approach, high quality people know intuitively how to produce high-quality software. Certainly, high quality people will produce relatively higher quality output. But if that's all that is required, then the best software organizations with arguably some of the best software professionals, wouldn't need process and wouldn't experience the common problems associated with software quality and productivity. Certainly, smaller projects require smaller processes, but even the smallest projects involving more than a single person could benefit from rudimentary processes to facilitate communication and ensure the synchronization of work [1].

A well-defined and tailored process will actually enhance creativity and free employees to concentrate on the details of the projects rather than the process [5, 9]. By freeing the most talented software professionals from the predicaments created by others (by avoiding the problems in the first place), process not only enhances creativity, but maximizes the productivity of the individuals and increases predictability of the work products [1]. Well defined and proportionate software processes can and do enable a person to do their job better, and not just prevent them from doing harm [11].

Further, research suggests that employee morale improves along with a well-managed SPI program. Boeing's Space Transportation Systems organization made the journey from CMMI level 1 to level 5 and at the same time employee satisfaction increased significantly along the way. The increase in satisfaction led to increased motivation, innovative solutions, greatly improved attitudes, and an increased sense of trust between management and employees [12]. Brodman, et al, while assessing the return-on-investment among 35 companies of various sizes that implemented the CMM[1] in their organizations, made the observation that the most frequently cited benefit was the morale and confidence of the software development staff improved "significantly" [13]. Additionally, there is anecdotal evidence that SPI increases employee ownership of the quality system and employees appreciate the strengthened support infrastructure [11].

Fundamentally, SPI provides a framework so continuous controlled improvement can be established to better support an organization's business objectives [1, 8]. Organizations seek to pursue SPI for a number of reasons [1, 5, 8].

- A desire to reduce wasted time and/or effort
- A desire to stabilize current processes to ensure repeatability
- To ensure that failures happen only once
- To remedy dissatisfaction with the status quo
- To rein in cost estimates and meet schedule commitments
- To enhance predictability

Crisis can be a great motivator for organizational change, but SPI can also be implemented by forward-thinking organizations in anticipation of growth or to take advantage of current software best-practices represented by the chosen reference process model.

The key characteristic of SPI is that improvement is both incremental and continuous. Processes are improved over time and experience is fed back into the improvement meta-process in a cyclical manner. Feedback occurs through the use of project retrospectives and regular process assessments. As an organization matures, its processes also mature as industry best practices are integrated into an organization's *modus operandi*. In this way, a mature software organization is dynamic, evolving, and improvement is continual.

2.2 SPI for Smaller Software Organizations

Smaller software organizations may have particular concerns that existing process models won't provide the flexibility required to address the great diversity of projects encountered. Though many organizations feel their particular project and process challenges are unique, software organizations all confront similar problems [1]. On one level, all software projects are different. But in many fundamental ways, all software projects are the same [5]. SPI must be tailored to match each organization's unique requirements, but at the same time leverage the field-tested best-practices [14].

Smaller software organizations (those with 50 employees or less) face unique challenges when pursuing SPI initiatives. Software Engineering for this 90% isn't simply a "degenerate" form of what is practiced in larger organizations, but has its own set of problems and solutions [6]. The following issues have been identified as particular impediments to smaller software organizations in their pursuit of SPI [15,16,17,18]:

[1] CMM was the precursor to CMMI.

- Questionable relevance of reference process models
- Cost of SPI implementation
- The trade-off of SPI versus other needs
- Incompatible corporate culture
- Customers not supportive
- Reference model organizational structure conflicts with current structure
- Resource limitations for pursuing SPI
- Heavy reliance on software “heroes”
- Long-term investment in SPI is not understood or appreciated
- Employees are often not aware of basic principles of quality management
- Perceive SPI to be for larger organizations only
- Not convinced by SPI data from larger organizations
- Lack of in-house expertise in SPI
- Not aware of potential business benefits
- Forced to adopt customer process standards
- Employees “opt-out” of larger organizations to avoid “process”

Among the larger themes from this list of issues is the sensitivity of smaller software organizations to cost and resource issues. Return-on-investment (ROI) is of particular importance, if not paramount importance. ROI must be justified at each step of the way. Smaller software organizations absorb the fixed costs of SPI in a disproportionate manner compared to larger organizations. Thus on a percentage basis, SPI costs smaller organizations more [15].

The following are basic principles to consider when implementing a SPI program [1, 8].

- Major changes must start at the top
- Everyone must be involved
- Knowledge of current processes is essential
- Process change is “continuous”
- Software process change requires reinforcement
- SPI requires investment (long term)

Organizations that attempt to improve process from the “bottom-up” end up with “islands of excellence”, but lack the follow-through to propagate the best-practices throughout the organization [5]. This phenomenon is particularly prevalent in smaller software organizations that tend to address process issues at the project level. SPI encourages the broadening of the myopic tendency to make process decisions based on expediency rather than on a planned basis. As mentioned earlier, SPI programs in small software organizations must always keep in mind the potential ROI and expend limited resources that will provide the most benefit with the least risk [8].

To ensure SPI efforts do not impinge on organizational innovation and creativity, the focus of SPI efforts should be on project-specific issues, rather than process centered ones. Standards and procedures should concentrate on essential details, but leave plenty of flexibility to accommodate the project variations found in typical small software organizations. One approach is to specify two levels of standards and procedures. The base would consist of mandatory elements whereas the top layer would be a set of best-practices that are recommended, but not required [11]. It’s also important to recognize that there’s no single right way to do SPI. The goal of a SPI effort must meet the requirements of the organization pursuing the effort [19].

Software organizations should also keep in mind the key factors of a successful software process improvement program [15, 8, 20].

- Company-wide SPI emphasis / Goals are widely understood
- Customer support of SPI efforts
- Management business alignment with SPI / Management support

- Sub-contracting relationship with a larger company with a SPI focus
- Membership in a third-party process improvement organization
- Process related training
- Developer / technical staff involvement
- Maintenance of momentum
- Corporate cultural awareness of SPI
- Separation of process and product concerns (for a product company)
- Presence and empowerment of champions
- Frequent process assessments
- Visibility into the SPI process
- Compensated SPI responsibilities
- SPI staff well-respected in the organization

Obviously, not every organization will be able to conform to these success factors. Instead, the list is meant to provide practical guidance on the factors that have been observed among small software organizations with successful SPI programs. Besides the support of top-level management, one other factor deserves special emphasis because of its repeated emphasis in SPI literature. Training in software process and SPI is almost universally highlighted and its ROI in regards to SPI is always emphasized as highly favorable [1, 5, 15, 11, 8, 19].

3. Software Process Assessment

3.1 Introduction to Software Process Assessment

Software process assessments are the mechanism by which SPI is measured, evaluated, and opportunities for improvement are discovered. Assessments are not an *ad hoc* exercise and imply a process standard to be evaluated against [1]. Watts S. Humphrey in his seminal work, *Managing the Software Process*, describes three main objectives for software process assessments - to learn the organization, identify problems, and enroll opinion leaders in SPI [1]. Problems identified during the assessment process serve as the raw material for developing process improvement initiatives. Assessments can also serve an educational function by encouraging process-level discussion, continuous improvement orientation, and facilitating a forum for creative group thinking in regards to process [21]. In a way, assessments provide a ten-thousand foot view of how a software organization is performing in meeting its business objectives.

Watts S. Humphrey identifies five guiding principles for software process assessments [1]:

1. Need for a process model
2. Absolute confidentiality
3. Senior Management Involvement
4. Attitude of respect towards all participants
5. Action orientation

A process model provides the framework or skeleton to organize the assessment and process improvement efforts. Without a process model framework, assessment and improvement efforts can devolve into a battle of personal opinion and conjecture. A process model provides an “ideal” based on known best practices. The purpose of an assessment is to evaluate an organization against these proven best practices and identify initiatives with the greatest ROI potential.

Confidentiality ensures that assessment interview questions and questionnaires are answered openly and honestly without fear of retribution or blame. The assessment process should always start with the assurance of confidentiality and the assessor should carefully guard personally identifiable information

from disclosure (e.g. names and projects). Further, freeform answers to assessment questions should be sanitized and only reported in aggregate form.

Since the assessment process requires resources and SPI is only effective with long-term follow-through, management approval and support is important. This is especially true if one of the formal process frameworks such as CMMI is employed, which often require extensive training, outside consultants, and time from project members. One of the most effective ways to ensure management buy-in of the assessment and SPI efforts is to involve management from the beginning of the project. Managers are particularly concerned about efficiency and even skeptical management can be won-over if the focus of SPI is on increased productivity, improved quality, and improved customer satisfaction.

Some of the best and most creative ideas come from non-management software development staff. It's always surprising how aware individual software team members are about process problems and how forthcoming they can be if there is a spirit of mutual trust between the team and the assessor. It's therefore critical the entire assessment process be based upon mutual respect. Process assessment concerns more than just management and all team members will have valuable input.

The outputs of a software process assessment are recommendations for tangible and actionable SPI initiatives. The guiding principle should be action-orientation instead of conformance to a particular process model or framework. The action plan must also support the strategic business goals of the organization.

3.2 Traditional Software Process Assessment

Traditional software process assessments consist of three phases, preparation, assessment, and recommendations [1]. The preparation phase is where assessment goals are determined, the process model framework is chosen, management buy-in is achieved, opinion-leaders are engaged, outside consultants are hired, and assessment materials are prepared. The assessment phase is the execution period when the assessment team performs the actual process assessment. The assessment phase represents the period of maximum involvement from the organization. Finally, the recommendation phase consists of analyzing the assessment results, prioritizing process areas in need of improvement, and reporting on the results.

Software process assessments can be performed by one of several groups depending on the level of formality required and the purpose of the assessment. For example, assessments performed for external compliance or certifications are usually performed by outside consultants. However, assessments performed for internal process improvement needs can be performed by a champion within the organization. Larger companies might have a process or quality group that performs process assessments. Typically, process assessments are performed by a cross-disciplinary group of key individuals such as management, team leads, SQA, and so forth. In the author's opinion, the more organic and indigenous the assessment team, the higher the probability of success.

Assessment questionnaires are created based on the key process areas (KPAs) enumerated in the reference process model. A scoring system is used to rate the completeness of the organization's deployment of each KPA. Each appraisal score is then aggregated into a summary score to generate an overall process maturity appraisal score (e.g. CMMI maturity level). Project artifacts are collected to provide verifiable objective evidence (VOE) to support the organization's appraisal. The process questionnaire results combined with the assessment scores are then reported back the organization along with process improvement recommendations.

Process assessments are typically performed on a periodic basis every 6-24 months. Process improvement takes time and assessments require resources to perform. The effort must be balanced against the potential benefit. As an example, the average time for an organization to move from CMMI level 2 to level 3 is 20 months for 2011 [22]. Thus, performing a process assessment every 3 months for a CMMI level 2 organization may not be an efficient use of resources.

3.3 CMMI-DEV and SCAMPI Appraisals

The Standard CMMI Appraisal Method for Process Improvement (SCAMPI) is the recommended software process appraisal method of the Software Engineering Institute (SEI). The purpose of a SCAMPI appraisal is to evaluate an organization's strengths and weaknesses against the CMMI process framework. According to the SEI, SCAMPI appraisals are conducted for the following reasons [23].

1. To compare an organization's process performance to CMMI best practices.
2. To provide a means of external certification to outside customers and suppliers.
3. To fulfill contractual obligations to customers.

As with all software process assessment frameworks, CMMI-DEV is based on Software Engineering best practices as defined by the SEI in association with industry and government. KPAs for CMMI-DEV are listed in Table 1 [23].

Process Area	***Category***	***Maturity Level***
Causal Analysis and Resolution (CAR)	Support	5
Configuration Management (CM)	Support	2
Decision Analysis and Resolution (DAR)	Support	3
Integrated Project Management (IPM)	Project Management	3
Measurement and Analysis (MA)	Support	2
Organizational Process Definition (OPD)	Process Management	3
Organizational Process Focus (OPF)	Process Management	3
Organizational Performance Management (OPM)	Process Management	5
Organizational Process Performance (OPP)	Process Management	4
Organizational Training (OT)	Process Management	3
Product Integration (PI)	Engineering	3
Project Monitoring and Control (PMC)	Project Management	2
Project Planning (PP)	Project Management	2
Process and Product Quality Assurance (PPQA)	Support	2
Quantitative Project Management (QPM)	Project Management	4
Requirements Development (RD)	Engineering	3
Requirements Management (REQM)	Project Management	2
Risk Management (RSKM)	Project Management	3
Supplier Agreement Management (SAM)	Project Management	2
Technical Solution (TS)	Engineering	3
Validation (VAL)	Engineering	3
Verification (VER)	Engineering	3

Table 1 - CMMI-DEV Key Process Areas

Each CMMI-DEV KPA is broken down further into sub-processes, which are evaluated separately during an assessment. CMMI uses a 5-level maturity scale (4-level for continuous representation) [23].

1 – Initial
2 – Managed
3 – Defined
4 – Quantitatively Managed
5 – Optimizing

Between 2006 and 2011, only about 7% of all organizations reporting to the SEI had been assessed at level 4 or 5 [22]. This suggests most software organizations are not motivated to pursue maturity levels

beyond 3 (managed). This supports concerns expressed by Karl Wiegers of Process Impact concerning the expected benefit of CMMI levels 4 and 5 versus the effort to achieve them [24].

One thing to note about the CMMI-DEV is the emphasis on process and project management KPAs. Many of the CMMI KPAs can rightly be considered meta-processes. CMMI specifies the “what”, but not the “how” of the processes and associated practices. As a result, there is considerable flexibility in the implementation of the process areas.

Due to the prestige of the SEI and ubiquity of the CMMI process framework throughout industry and government, CMMI process maturity ratings are influential when evaluating customers and suppliers. This “process report card” aspect to the CMMI allows business and government to evaluate the process maturity (and indirectly quality) of its vendors. Likewise, a vendor can use a high CMMI appraisal level to promote its process maturity. Of the1333 SCAMPI Class A appraisals reported to the SEI in 2011, 633 were reported by software organizations in China and India versus only 270 reported from all U.S. software organizations [22]. The large number of offshore software development organizations in both China and India likely account for most of the disparity.

3.4 Where the CMMI and SCAMPI Appraisals Fall Short

The CMM and later the CMMI were developed by the SEI, which is largely funded by grants from the U.S. Department of Defense (DoD). Given this lineage, CMMI has traditionally been used by the DoD to evaluate suppliers. For many software organizations doing business with the DoD, CMMI appraisals and a minimal maturity level are required for contract compliance. For these organizations, large and small, there are strategic business reasons to adopt a formal CMMI approach. Large software organizations (i.e. hundreds of developers) may also find the rigor and formality of a full CMMI approach compelling. For large organizations, the cost and resources required to fully implement a CMMI SPI program are easier to absorb.

What about the other 90% of software organizations with less than 50 employees? For many smaller software organizations, the resources required to pursue a CMMI-based SPI program are often prohibitive. A SCAMPI appraisal team consists of at least four people and one of the four must be a certified lead appraiser. Lead appraisers must take at least 3 CMMI training courses, pass a number of tests, and have previous experience with SCAMPI appraisals. Other members of the appraisal team must take at least one CMMI training class. The expense and effort is considerable, which is why most software organizations hire external consultants to perform SCAMPI appraisals. Lead Appraisers typically charge between $1,500 and $2,500 per day [25]. It’s easy to see how costs for a SCAMPI appraisal can approach $50,000-100,000. For many smaller software organizations, the potential ROI simply does not justify the cost.

Even if the CMMI is only utilized for the process model without performing SCAMPI appraisals, CMMI may not be the best choice for reasons beyond cost. Of the 22 KPAs in the CMMI-DEV profile, 12 are focused on project and process management activities. Only 5 of the 22 KPAs are categorized as engineering process areas. Yet, most software organizations pursue SPI to reduce waste, reduce occurrence of failure, improve estimates, enhance predictability, and produce higher quality outputs. The CMMI process areas are weighted toward practices that don’t necessarily support these goals. Ultimately, the CMMI can help develop an elaborate process infrastructure, but says surprisingly little about how to develop better software.

The CMMI provides little prescriptive guidance on how to implement the required processes. For example, CMMI-DEV specifies that peer reviews and code reviews are to be performed, documented, and metrics collected. There’s only generic guidance of the style or rigor of the reviews, whether this includes such best practices as static code analysis, how often reviews should be performed, or which work products are to be reviewed. The net effect is that organizations can develop elaborate software processes based on the CMMI, but still not have a handle on how effective the processes are in regards to overall software quality (e.g. defect removal efficiency) or productivity.

What's needed is a lightweight and adaptable SPI and assessment approach that leverages established industry software best practice research and provides specific and tangible process guidance. Considerable research has been done on the relative effectiveness and productivity impact of most common Software Engineering best practices [26]. Such an approach would not be usable as an external assessment or certification vehicle like the CMMI. But, most software organizations (i.e. the other 90%), are primarily interested in SPI for internal improvement and only require a process framework to provide a rough guide for long-term continuous SPI. The standardized and externally-facing aspects of the CMMI are of little value in this scenario.

4. Lightweight Process Assessments

4.1 Software Engineering Best Practices

Software assessments must be performed in relation to known best practices to be effective. Without an ideal to compare against, a software organization won't know where it falls short. Ideally, these best practices are based on sound Software Engineering research. The most common software process frameworks are very complex and can be difficult to implement. Yet, most software organizations don't have internal expertise in SPI. Unless an organization is willing to expend the resources to hire outside consultants, another approach is required.

Capers Jones and his company, Software Productivity Research, have been conducting Software Engineering research for over 25 years. Capers Jones' recently published book, *Software Engineering Best Practices*, contains compiled data from 675 companies in 24 countries and 13,500 projects [26]. The data is analyzed with three dimensions, code size, type of software, and development activity. Each practice/process is ranked based on defect removal efficiency (DRE)[2] and productivity. 200 practices are scored and ranked. Scores range from 10.00 to -10.00, with negative scores indicating practices that do more harm than good. The following table shows the first 50 best practices overall.

Rank	Methodology, Practice, Result	Score
1.	Reusability (> 85% zero-defect materials)	9.65
2.	Defect potentials < 3.00 per function point	9.35
3.	Defect removal efficiency > 95%	9.32
4.	Personal Software Process (PSP)	9.25
5.	Team Software Process (TSP)	9.18
6.	Automated static analysis	9.17
7.	Inspections (code)	9.15
8.	Measurement of defect removal efficiency	9.08
9.	Hybrid (CMM + TSP/PSP + others)	9.06
10.	Reusable feature certification	9.00
11.	Reusable feature change controls	9.00
12.	Reusable feature recall method	9.00
13.	Reusable feature warranties	9.00
14.	Reusable source code (zero defect)	9.00
15.	Early estimates of defect potentials	8.83
16.	Object-oriented (OO) development	8.83
17.	Automated security testing	8.58
18.	Measurement of bad-fix injections	8.50
19.	Reusable test cases (zero defect)	8.50
20.	Formal security analysis	8.43
21.	Agile development	8.41
22.	Inspections (requirements)	8.40

[2] DRE = Defects Discovered Before Release / Total Discovered Defects Before and After Release.

23.	Time boxing	8.38
24.	Activity-based productivity measures	8.33
25.	Reusable designs (scalable)	8.33
26.	Formal risk management	8.27
27.	Automated defect tracking tools	8.17
28.	Measurement of defect origins	8.17
29.	Benchmarks against industry data	8.15
30.	Function point analysis (high speed)	8.15
31.	Formal progress reports (weekly)	8.06
32.	Formal measurement programs	8.00
33.	Reusable architecture (scalable)	8.00
34.	Inspections (design)	7.94
35.	Lean Six Sigma	7.94
36.	Six Sigma for software	7.94
37.	Automated cost-estimating tools	7.92
38.	Automated maintenance workbenches	7.90
39.	Formal cost-tracking reports	7.89
40.	Formal test plans	7.81
41.	Automated unit testing	7.75
42.	Automated sizing tools (function points)	7.73
43.	Scrum session (daily)	7.70
44.	Automated configuration control	7.69
45.	Reusable requirements (scalable)	7.67
46.	Automated project management tools	7.63
47.	Formal requirements analysis	7.63
48.	Data mining for business rule extraction	7.60
49.	Function point analysis (pattern matches)	7.58
50.	High-level languages (current)	7.53

Table 2 - Top 50 Practices from Software Engineering Best Practices

The reader should immediately notice how specific and tangible many of the practices are. Unlike the CMMI, which is very abstract and management focused, SPR's best practices are focused on practices and processes that improve early defect detection, such as Team Software Process, automated static analysis, and code inspections. The CMMI only refers to these practices indirectly, if at all. For software organizations looking to improve software quality as efficiently as possible, seeking to improve DRE is a good first step.

The best practices presented here and in Capers Jones' book are meant to be a starting point and not all practices will be relevant in every context. Different organizations will have different strengths and weaknesses. The history and experience of each organization will also vary. There may be practices not on the list that an organization would like to address due to familiarity or an existing partial implementation. Since the goal is to improve software processes internally, there is considerable flexibility and opportunity for customization. To maximize ROI, it's essential to focus on process improvement generally and not get distracted by the allure of conformity to a process model or an arbitrary set of practices.

4.2 The Assessment Process

A software process assessment should be managed as a project with the requisite planning. The high-level assessment tasks are enumerated in Figure 1. As with all projects, the assessment process begins with a charter to ensure buy-in from key stakeholders. The charter need not be particularly formal, but due to the time and effort required to perform an assessment, at least tacit management approval is needed. The charter also establishes the general goals and nature of the assessment. Issues such as the

assessment scope, level of formality, the composition of the assessment team, rough timeframe, and the disposition of the assessment outputs should be defined early in the process.

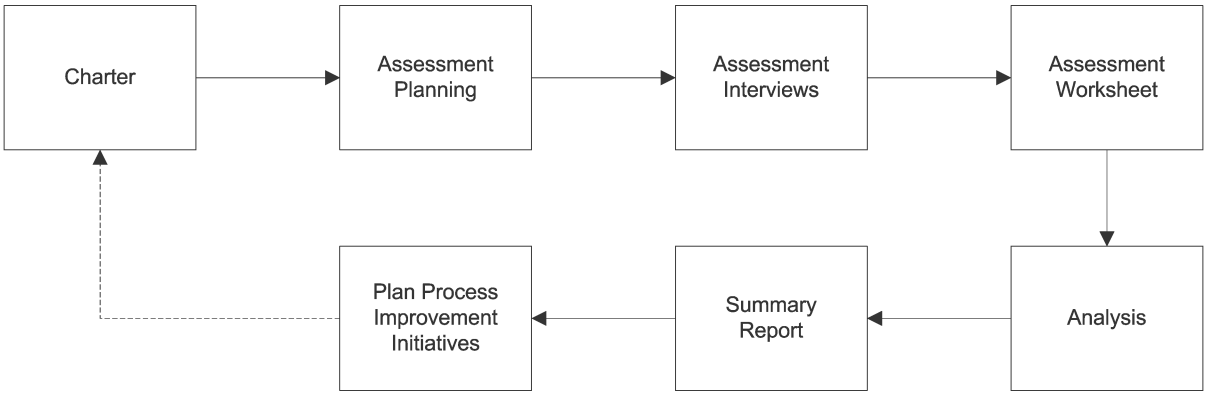


Figure 1 - Software Process Assessment Process Diagram

Assessment planning establishes the roadmap for the assessment process and sets expectations. The primary goal of the planning phase is scheduling assessment meetings and interviews. The assessment plan may be as informal as an email thread or as formal as a complete project plan. It's recommended that the assessment plan not be too prescriptive to allow the process to evolve dynamically. For instance, it is often necessary to engage additional resources based on new information discovered during the course of the assessment process.

There are two aspects to a software process assessment – qualitative and quantitative. The qualitative aspect captures the non-quantifiable input to the assessment and represents the "soft side" of the assessment. Qualitative data is generally gathered from assessment interviews performed one-on-one with key members of the organization. The number of interview subjects will vary from organization to organization. But, generally, interviews with a quarter to half of an organization provide adequate input to the process. Interviews are typically about an hour in duration, but definitely not longer than two hours. Interview questions are customized for an organization based on the broad process areas common to most process frameworks (e.g. Management Processes, Organizational Processes, Engineering Processes, etc.). The qualitative data is captured "raw" and archived for later analysis. As stated previously, the answers to the interview questions should be sanitized for personally identifiable information to ensure the anonymity the participants. The appendix to this paper includes an actual assessment survey from a software process assessment performed by the author in 2005.

The quantitative aspect of an assessment is a rough attempt to aggregate assessment results into a single number or set of numbers. Figure 2 is an example software process assessment worksheet for a single software organization.[3] Assessment results are summarized by process area, which enables analysis by related categories of best practices. The best practices in this worksheet were chosen based on Capers Jones' research published in *Software Engineering Best Practices* discussed in the previous section [26].

The implementation of each best practice is rated on a simple 3-level scale.

1. Currently doing nothing or very little
2. Partial implementation, but opportunity for improvement
3. Fully Implemented

This scale, which is a simplified version of the CMMI scale, makes assessment very easy and reduces ambiguity overall. Scores for individual best practices are then averaged together for each process area and the process areas are averaged together to produce an overall assessment rating. Individual

[3] Document templates are available from the author upon request.

organizations can modify this approach by using a more graduated scale and/or weighting certain process areas over others.

By nature, quantitative measures are inexact and abstract. However, in the aggregate, quantitative measures provide valuable trending information to guide SPI efforts over time. Quantitative measures are meant to be used as relative measures and not as absolute measures. An organization that is increasing in process maturity should see its assessment score rise over time. The author has also observed a strong correlation between the implementation of proven best practices and software quality.

Software Best Practices Assessment Worksheet

Rating Key: 1 - Currently Doing Nothing or Very Little; 2 - Partial Implementation, but Opportunity for Improvement; 3 - Fully Implemented

Best Practice	Sep-10	Dec-10	Mar-11	Jun-11	Sep-11	Dec-11
Software Process/Project Management	**2.33**	**2.50**	**2.50**			
Regular Progress Reports/Standup Meetings	3	3	3			
Customer Visits/AE Interviews/VOC Data/Trip Reports	2	3	3			
Robust Configuration Management of All Deliverables	3	3	3			
Software Process Explicitly Defined and Documented	2	2	2			
Clear Exit Criteria/Checklists for All Project Milestones	2	2	2			
Formal Estimation Discipline	2	2	2			
Software Requirements	**2.29**	**2.43**	**2.71**			
Requirements Inspections/Reviews	2	2	3			
Formal Requirements Tracking/Product Backlog	3	3	3			
Measurement of Requirements Changes	1	2	2			
Changes Managed via Change Control Board (CCB)	3	3	3			
Non-Functional Requirements Explicitly Defined	1	1	2			
Usability is a Core Part of the Requirements Process	3	3	3			
Use Case/User Story-Driven Requirements	3	3	3			
Software Design/Architecture	**1.50**	**2.00**	**2.17**			
Design Inspections/Reviews	2	2	2			
Formal Architecture Documentation	1	2	2			
Reusable Architecture	2	3	3			
Architecture Follows Solid Design Principles	2	2	2			
Design Documents Updated Throughout the Lifecycle	1	1	2			
Design Documentation is Traceable to Requirements	1	2	2			
Software Construction	**1.78**	**2.33**	**2.33**			
Software Reuse	2	3	3			
Source Code Control (e.g. ClearCase)	3	3	3			
Static Code Analysis	1	2	2			
Automated Defect Tracking (e.g. ClearQuest)	3	3	3			
Peer Code Inspections	1	2	2			
Agile/Scrum Development/Incremental Delivery	3	3	3			
Continuous Integration	1	2	2			
Formal Coding Standards	1	2	2			
Preference for COTS Components versus Rolling Your Own	1	1	1			
Software Testing	**2.00**	**2.29**	**2.29**			
Reusable Test Cases	2	2	2			
Formal Test Plan (reviewed)	2	2	2			
Automated Testing (Unit, Regression, and/or Integration)	2	2	2			
Expert User Testing/Lighthouse Customers/Beta Testing	2	2	3			
Integration Testing on Actual Target Hardware Early	2	3	2			
Test Case Traceability to Requirements / User Stories	2	3	3			
Unit Testing/Test Driven Development	2	2	2			
Software Quality Assurance (SQA)	**1.75**	**2.50**	**2.75**			
Formal SQA Plan (reviewed and approved)	1	1	2			
Separate/Independent SQA Function	3	3	3			
Formal Risk Management	2	3	3			
Defect Removal Efficiency is Measured	1	3	3			
Overall Assessment Rating	**1.94**	**2.34**	**2.46**			

Figure 2 - Example Software Assessment Worksheet

4.3 Assessment Analysis

Once the software process assessment is complete, the raw results must be analyzed to identify actionable process improvement initiatives. There are a number of ways to evaluate assessment results and generally it's good to start with the goals and priorities laid out in the assessment charter. The qualitative information gathered from the assessment interviews is very important for setting the context and providing background for the assessment analysis. Common and recurring themes will often arise and should be carefully considered. The best improvement ideas often come from the assessment interviews and the assessment team should seek to keep an open mind when analyzing the results. The assessment interviews also provide the bulk of the narrative for the software process assessment report.

One method of analyzing and prioritizing best practices based on effectiveness and effort is an impact matrix, shown in Figure 3. The vertical axis is an estimate of the effort required and the horizontal axis is the defect removal efficiency based on Capers Jones research. The practices in the lower-right quadrant are considered low-hanging fruit due to the relatively low effort required for implementation and the high defect removal efficiency potential. Whereas the defect removal efficiency of a practice is objectively measured based on research, the implementation effort will often vary between organizations depending on history, process maturity, and existing capabilities. In general, practices such as software reuse, peer review, static code analysis, formal risk management, and coding standards provide a "big bang for the buck" regarding overall software quality improvement.

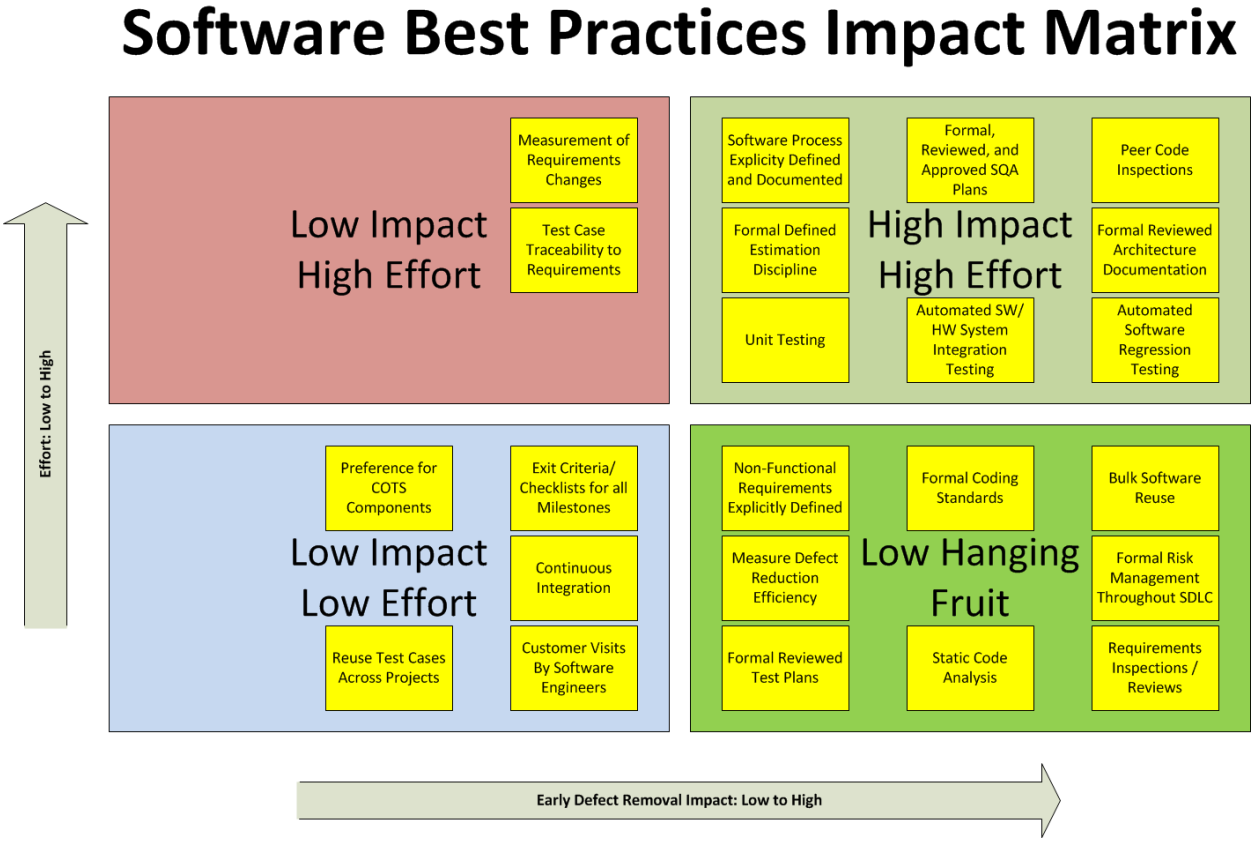


Figure 3 - Impact Matrix Demonstrating One Method for Analyzing Possible SPI initiatives

The typical deliverables of a software process assessment are as follows.

1. Description of the current state of software quality - It's important to be sensitive to an organization's historical context and prior process improvement efforts.

2. An analysis of organizational strengths and weaknesses – It's best to lead with the strengths, but an organization will usually accept a carefully considered presentation of its weaknesses.
3. Hard data and metrics – In addition to the software process assessment worksheet, other metrics can be useful to paint a full picture (e.g. defect removal rates, project schedule delays, customer satisfaction survey results, etc.).
4. A methodology overview – How did the assessment team reach the conclusions and recommendations being proposed?
5. Description of the future state of software quality – It's essential to explain the path from the present state to the future state is an incremental one.
6. A list of software process improvement initiatives – The number of initiatives must be realistically considered. In general, the list should not be longer than can be achieved in 12 months or less.

There are a number of ways to present the software process assessment results, but typically an assessment report will be issued followed up by a presentation of the results. Some organizations will also integrate training for specific practices with the presentation of the assessment results. Each organization will ultimately have to decide the most effective way to communicate and initiate the proposed changes. At a minimum, it's essential to plan and schedule the proposed initiatives.

Figure 4 shows an example software process improvement backlog. Each initiative is broken down into subtasks with task ownership identified, a tentative target date, and current status. Figure 5 shows the same tasks presented on a visual Kanban board for display in a public area such as a shared hallway.

Software Process Improvement Backlog				
Initiative / Task	**Owner**	**Target Date**	**Status**	**Notes**
Resharper		**End of Q1 2011**	**In progress**	
Acquire Resharper licenses.	Steve	12/2/2010	Complete	
Install Resharper on each workstation.	Team	12/9/2010	Complete	
Provide additional Resharper training for the software team (if needed).	Steve	Q1 2011	Complete	not needed
Create a Resharper code style profile for the DAPL C# coding standard.	Bob	Q1 2011	Not started	
Peer Reviews / Code Inspections		**End of Q1 2011**	**In progress**	
Decide on a Peer Review tool.	Bob	10/27/2010	Complete	
Acquire PeerReview Complete	Bob and Jill	12/16/2010	Complete	
Establish a permanent server hosting environment for PeerReview Complete.	Bob	1/27/2011	Complete	
Install and Configure PeerReview Complete	Bob	2/17/2011	Not started	
Unit Testing		**End of Q1 2011**	**In progress**	
Retrofit existing Apollo unit tests into Microsoft Test framework.	Steve	11/18/2010	Complete	
Create a DAPL unit test standards document.	Bob	1/27/2011	In progress	
Setup initial unit test project structure in the Apollo .Net solution.	Steve or Bob	1/27/2011	In progress	
Revise sprint completion criteria checklist to specify unit testing requirements.	Bob, Steve, or Joe	Q1 2011	Not started	
Research unit test alternatives for embedded C/C++ code.	Bob	11/25/2010	Complete	FlexeLint is the best option.
Coding Standards		**End of Q1 2011**	**In progress**	
Investigate and adopt a C# / .Net coding standard.	Bob and Steve	2/10/2011	In progress	Current proposal is to use the *.Net Framework Design Guidelines* book and FxCop standards.
Document new coding standard.	Bob	2/24/2011	Not started	If we use the *.Net Framework Design Guidelines* coding standards, we'll just purchase multiple copies of the book.
Provide additional review and training of the coding standard as needed.	Bob and/or Steve	Q1 2011	Not started	This may include training on defensive programming and common gotchas.
C# Static Code Analysis (FxCop)		**End of Q2 2011**	**Not started**	
Determine the policies and guidelines for applying static code analysis to the existing code base (e.g. which projects to analyze, how to manage findings).	Bob, Steve, and Joe	4/13/2011	Not started	
Perform initial static code analysis, tune exceptions, and generate report.	Bob	5/18/2011	Not started	
Integrate static code analysis into build process.	Bob, Steve, Pam	Q2 2011	Not started	
Manage static code analysis findings on an ongoing basis.	Bob	Ongoing	Not started	
Continuous Integration		**End of Q2 2011**	**Not started**	
Acquire a server to host the continuous integration environment.	Bob	Q1 2011	In progress	We will be using CruiseControl for continuous integration.
Determine if ClearCase will work for continuous integration or if we need to use Subversion.	Bob	Q1 2011	Not started	
Install and configure continuous integration environment.	Bob	Q2 2011	Not started	
Provide continuous integration training for the software team.	Steve	Q2 2011	Not started	
Integrate continuous integration environment into day-to-day development.	Team	Q2 2011	Not started	

Figure 4 - Example Software Process Improvement Backlog



Figure 5 - Example Software Process Improvement Kanban Board

5. Conclusion and Lessons Learned

The SPI process is not a one-time endeavor. Change must be incremental and continuous to have a lasting impact. The basic continuous improvement framework is based on the Plan-Do-Check-Act (PDCA) model, otherwise known as the Deming Cycle or Schewhart Cycle. The PDCA model is the fundamental basis for all continuous improvement. It's expected that most organizations will customize PDCA to meet their unique needs.

In a software organization committed to continuous improvement, software process assessments should be performed every year or two and new process improvement initiatives should be regularly proposed and implemented, even if it's just one or two initiatives at a time. SPI is ultimately an investment that is amortized over time. As such, SPI introduces overhead and consumes organizational resources. Each organization will have to determine the optimum rate of change.

To a certain extent, all software organizations are complacent and driving change is essentially hard. One should expect resistance to SPI, but this should not be discouraging. If SPI were easy, then every software organization would always deliver high quality software on time. There are a number of ways to mitigate change resistance.

- Aggressively pursue management support.
- Enlist the organizational thought leaders in the SPI process.
- Utilize pilot initiatives limited to one project team or a subset of the organization.
- Track quality metrics to illustrate both improvement opportunities and SPI successes.
- Always propose initiatives with the highest ROI potential first.

The assessment process itself should also be continuously improved. Learn from your assessment experiences. Make use of post-assessment retrospectives to collect feedback on what went wrong, what went right, and ideas for improving the assessment process in the future. Between assessments, look for ways to improve the SPI process overall (e.g. keep a SPI notebook). Promote software quality and process maturity continuously.

In the end, SPI is as much of an art as it is a science and art must be experienced. The following is a list of hard-learned lessons based on the author's experience. Your mileage may vary.

1. If there's one thing the reader should take away is that SPI does not need to be a heavyweight undertaking. By focusing on continuously improving high ROI processes a few practices at a time, rapid and permanent change is very attainable.
2. Be aware of the Hawthorne Effect, which says that simply observing an organization often changes the behavior of the organization. Utilize this phenomenon by regularly shining a light on quality problems and process areas that need improvement.
3. SPI works best if change develops from within a team or an organization and is not imposed from the outside. Stand-alone quality groups rarely are able to effect the long-term change required to fundamentally improve software quality.
4. Very often, the best SPI feedback and ideas come from front-line employees who perform the actual productive work (e.g. developers, project managers, SQA staff, etc.).
5. It's almost always best to implement the best practices with the highest potential ROI and then heavily leverage early successes to motivate further improvement and change.
6. Long-lasting and transformative SPI is essentially cultural change. Changing the culture of an organization requires concerted effort and takes considerable time. Persevere, but be patient.
7. Crisis can be a great opportunity to implement change into an organization. If your software organization is resistant to change, be ready to seize crisis opportunities.
8. Software productivity and quality go hand in hand. However, quality always leads productivity. When a software organization fails to deliver on time, quality is almost always at fault.
9. It's essential to be realistic about what can be expected regarding the level and pace of change. Every organization has its own velocity at which change can be absorbed. Know this velocity.
10. Management support for SPI is very important. But, SPI can also be pursued "under the radar," especially if done organically at the team level.
11. Most successful SPI efforts are driven by a process champion who is an integral and respected member of a team or an organization.
12. Carefully chosen metrics can be very insightful and motivational. Nothing convinces management like solid numbers. Leverage metrics to drive SPI, if possible.
13. Ideally, SPI initiatives should be incentivized by management.
14. Ironically, resistance to change is often greatest from managers and team leads. It's up to the process improvement champion to convince management that SPI improves an organization's performance in ways relevant to them such as improving productivity, quality, and the organization's ability to innovate.

References

[1] W. Humphrey. Managing the Software Process, Addison-Wesley Reading Mass., 1989.

[2] Software Engineering Institute, Carnegie Mellon University, http://www.sei.cmu.edu/cmmi/.

[3] Available at http://www.iso.org/iso/search.htm?qt=15504&published=on&active_tab=standards.

[4] M.E. Fayad, M. Laitinen and R.P. Ward, "Software Engineering in the Small," Communications of ACM, vol. 43, no. 3, March 2000, pp. 115-118.

[5] M. Paulk, "Using the software CMM in small organizations," in Joint Proceedings of the Pacific Northwest Software Quality Conference and the Eighth International Conference on Software Quality, Portland. 1998, pp. 350–361.

[6] M. Laitinen, M. Fayad, R. Ward, "Guest Editors' Introduction: Software Engineering in the Small," IEEE Software, vol. 17, no. 5, September 2000, pp. 75-77.

[7] A.P. Cater-Steel, "Low-rigour, Rapid Software Process Assessments for Small Software Development Firms," in Proceedings of the Australian Software Engineering Conference, 2004.

[8] SPIRE Project Team, The SPIRE handbook; Better, Faster, Cheaper Software Development in Small Organizations, Centre for Software Engineering, Dublin, 1998.

[9] J. Bach, "Enough about process; What we need are heroes," IEEE Software. Vol. 12, no. 2, Feb. 1995, pp. 96-98.

[10] T. DeMarco, T. Lister, Peopleware: Productive Projects and Teams. New York: Dorset House Publishing Co.,1999.

[11] D.P. Kelly, B. Culleton, "Process Improvement for Small Organizations," IEEE Computer, vol. 32, no. 10, October 1999, pp. 41-47.

[12] G. Yamamura, "Head to Head: Process Improvement Satisfies Employees," IEEE Software, vol.16, no. 5, Sept./Oct. 1999, pp.83-85.

[13] J. Brodman and D. Johnson, "Return on Investment from Software Process Improvement as Measured by U.S. Industry". Crosstalk, Vol. 9, no.4, April 1996, pp.23-29.

[14] M.E. Fayad, M. Laitinen (1998) "Process Assessment Considered Wasteful," Communications of the ACM, November 1997, vol. 40, no 11, pp.125-128.

[15] J. Brodman and D.L. Johnson, "What Small Businesses and Small Organizations Say about the CMM" Proc. 20th Int'l Conf. Software Eng., IEEE CS Press, 1994, pp. 331-340.

[16] P. Grunbacher, "A Software Assessment Process for Small Software Enterprises," in Proceedings of the 23rd EUROMICRO Conference '97 New Frontiers of Information Technology, IEEE CS Press, 1997, pp. 123-128.

[17] G. Chroust, F. Stallinger, "Software Process Capability in Europe: A Survey," presented at EUROMICRO 99 – European Software Day, Milan, Italy, 1999.

[18] S.M. Garcia, "Thoughts on Applying CMMI in Small Settings," 2005, http://www.sei.cmu.edu/cmmi/adoption/pdf/garcia-thoughts.pdf.

[19] T. Pyzdek, "To Improve your Process: Keep it Simple," IEEE Software, vol. 9, no. 9, September, 1992, pp. 112-113.

[20] F. Guerrero and Y. Eterovic, "Adopting the SW-CMM in a Small IT Organization," IEEE Software, vol. 21, no. 4, July/August 2004, pp. 29-35.

[21] K. Wiegers, "Software Process Improvement: Ten Traps to Avoid," SD Times, May 1996.

[22] Software Engineering Institute, "CMMI for SCAMPI Class A Appraisal Results 2011 End-Year Update," March 2012, http://www.sei.cmu.edu/cmmi/why/profiles/upload/2012MarV3CMMI.pdf.

[23] Software Engineering Institute, "CMMI for Development, Version 1.3", November 2010, http://www.sei.cmu.edu/reports/10tr033.pdf.

[24] K. Wiegers, Creating a Software Engineering Culture, New York, NY: Dorset House Publishing, 1996.

[25] Entinex, Inc. (2011, October 9). *CMMIFAQ* [Online]. Available: http://www.cmmifaq.info/

[26] C. Jones, Software Engineering Best Practices, New York: McGraw-Hill, 2010.

Appendix - Sample Software Process Assessment Questionnaire

The following is a sample software process assessment questionnaire based on the ISO 15504 Software Process Improvement and Capability dEtermination (SPICE) process framework. The questionnaire was used by the author for a process assessment performed in 2005. Each interview took 1-2 hours to complete.

Management Processes

1) What is your opinion of the company's management team? Do you feel the management style in place is effective?

2) What would you change about the company's management style (if you could)?

3) What is your opinion of the company's project management? Do you feel that projects are well managed and executed? What changes / improvements would you like to see?

4) What is the company's basic project management approach? Is there a repeatable project management standard in place?

5) How is project progress reported and how often? How is a project monitored for acceptable progress?

6) Are there instances or generally areas where the existing project management approach may not be adequate?

7) What does "quality management" mean to you? Do you think the company has a quality focus? How is quality managed here?

8) How is risk managed in a typical project? What's the typical approach when a risk is realized and becomes a problem?

Organizational Processes

1) Do you feel the company relies on exceptional people (i.e. heroes)? Is there any concern that we may not be able to continue hiring heroes?

2) What role does training have at the company? Would more or less training be desirable?

1) How does the company typically illicit *customer* requirements? Is it always done in the form of and RFP / Proposal? What are the typical constraints placed on a project by a customer? Does the company use standard document templates for proposals?

2) Describe the process for initial customer requirements elicitation? Who's involved and how are stakeholders identified? What's the process for ongoing customer requirements changes?

4) Once coding actually commences, how is the actual code produced verified against the requirements and design?

8) What does the term "validation" mean to you? What validation processes does the company have in place (as far as you know)?

9) At what points in the software process do "joint reviews" occur? In your opinion, are they valuable?

10) In your opinion, do you think internal formal code and design reviews would be valuable?

11) How are problems typically resolved? Is there a formal process? How do contractual obligations play into the problem resolution process?

General Questions

1) What effect does the company's desire for follow-on work effect its software process discipline?

2) Are the company's projects growing in size and complexity?

3) Do you feel the company's existing software processes will scale as it grows?

4) How do you see the company's "culture" and how amenable do you think it is to a more formal software process approach?

5) Are there particular communication problems in the company? If so, when and where?

6) In your opinion / experience, what are the most pressing software process problem areas?

7) Finally, if you could change one thing about the company's approach to software development, project management, or culture, what would it be?

Development-Driven Testing: Ensuring Testing Meets the Needs of Software Developers

Tim Farley
tim.farley@gylity.com

Abstract

Testing to support the needs of software developers is fundamentally different from testing to support the needs of end users. While both end users (external customers) and software developers (internal customers) rely on testing to demonstrate the readiness of the product, what is meant by "product" and "ready" is completely different. For end users, the product is the final product and it is ready for use when it meets all requirements. For developers, the product is the software they are in the process of creating and it is ready when it supports further development.

Software Quality Assurance (SQA) and testing teams are often focused exclusively on ensuring final products are ready for external customer use. Such teams often lack either the capability or capacity to provide the testing services that internal software development customers require, beginning with a clear understanding of what testing benefits developers most. It is impossible to meet internal customer needs when those needs are unknown.

Development-Driven Testing (DDT) provides a framework for SQA and testing teams to understand and meet the needs of software developers. The approach assesses the interfaces between the SQA and development teams, the information exchanged between them and the timing of their interactions to determine whether they are adequate to support the needs of developers. Test development, documentation, execution and results reporting can then be optimized to improve that support. By ensuring that testing meets the unique needs of developers, SQA and testing teams can improve software development efficiency, shorten software development cycles and improve product quality.

This paper describes how to create a DDT strategy. It describes how to identify the unique needs of a development team and create a testing strategy to meet those needs by analyzing the interfaces, information and interactions between the teams. The paper concludes with a project walkthrough where adapting the testing strategy to meet developer needs led to improved product quality and development efficiency, and saved the company millions of dollars compared to legacy testing strategies.

Biography

Tim Farley has 25 years experience in software quality assurance. He has led teams working on products ranging from critical medical devices to iPhone apps. For the past 10 years, Tim has focused on test automation, tools and bringing consistent practices to diverse SQA and software development teams across large organizations. Tim last presented a paper at PNSQC in 1996 about creating "intranet" sites, which has since proven to be a pretty good idea.

Tim has degrees in Computer Science and Anthropology from Brown University.

Copyright Tim Farley 2012

1 Testing that Meets the Needs of Developers

Software developers have needs. Unfortunately, software quality assurance (SQA) engineers don't always know what those needs are. SQA organizations that understand how software developers work, what is important to them and how testing can help can have a significant positive impact on developer productivity, project duration and product quality. An SQA organization that understands the needs of developers can ensure that the right testing is done at the right time and in a way that makes further development efficient.

It's not enough, though, to understand this in general. It is critical for SQA teams to understand the exact needs of each development team for each specific project. What improved efficiency for one team may be an irritating distraction for another.

Development-Driven Testing (DDT) provides a framework for understanding the specific needs of software developers and assessing whether those needs are being met. It examines the interfaces through which SQA and development organizations communicate, the information that gets communicated, and the timing and participants involved in these interactions. It optimizes testing processes, tools and practices to ensure that developer needs are met.

Note: In this paper, SQA refers to software quality assurance and testing organizations. Software development organizations will be referred to as Development.

---

2 Creating a Development-Driven Testing Strategy

Creating a Development-Driven Testing strategy involves identifying the needs of Development, assessing whether SQA is currently capable of meeting those needs, and finally optimizing the testing tools, processes and practices to meet those needs.

2.1 Identifying Developer Needs

The first step in creating a DDT strategy is to understand the needs of the development team. These needs will be discovered through team discussions, observations and participation. There are four topics that can be examined first to get the process started.

- Efficiency – How much time do developers spend doing what they intended to be doing?
- Feedback – How long does it take to determine whether something is correct?
- Unique Needs – What are the special things this team is trying to do?
- Measurements – How are we doing?

2.1.1 Efficiency

Efficiency is the extent to which developers spend their time doing what they intended to be doing. Under ideal circumstances, this means never being distracted from creating working software. Unfortunately, there are many distractions that can prevent developers from doing this.

- Misunderstood Requirements
- Broken Builds
- Defects

Obviously, SQA shouldn't be creating distractions that make development less efficient, but sometimes distractions happen.

- Defect reports may be incomplete and require developers to gather missing information on their own.

- Tests may be run at the wrong time or on the wrong build.
- Test results may be reported that are difficult for developers to interpret.
- Tests may be incomplete and lead to product failures that are only discovered post-launch.

All of these can unintentionally distract developers, slow software development and reduce product quality.

There will also likely be services that SQA provides for developers that neither help nor hinder efficiency. For example, Development may be perfectly satisfied with manual testing. SQA tools, processes and practices that are merely sufficient for developers will likely become candidates for change when SQA begins optimizing. SQA should also understand where it is absent from the software development process and why. There may be opportunities for SQA to increase developer efficiency by performing completely new services. For example, SQA may only test integrated systems and not the individual components that make up that system. Testing components may provide new and unexpected efficiency improvements.

2.1.2 Feedback

Feedback is when we learn whether something we've created is actually correct. For example, when Development builds a new release, SQA tests that release to determine whether the software functions as expected. The feedback comes when test results are returned to Development. Likewise, when SQA creates a set of tests, the feedback comes when Development reviews the tests. Feedback always requires the creation of something new and the confirmation that it is correct.

Feedback ensures that projects are on track to deliver what they intended. The timeliness of the feedback and the response when it is received have an enormous impact on software development efficiency. For example, testing features as soon as the code is available provides timely feedback to developers, and immediately fixing any failures found maintains developer efficiency. Feedback from external customers post-release is always untimely feedback and fixing post-release problems distracts developers from current work. It can also distract SQA when post-release fixes require extensive regression testing to prove that the new code is better.

SQA needs to understand which Development deliverables require confirmation, what it takes to confirm the deliverable, and when that confirmation needs to be delivered. This requires dedicated cooperation and collaboration between the teams to ensure that the right feedback is delivered at the right time. For example, developers might want their code changes tested immediately after check-in. To provide timely feedback, SQA would need to know what changed, what tests are appropriate to qualify the changes, who needs to know the test results and when those results are needed.

Feedback content and format should always be appropriate for the intended audience and use. Feedback that is delivered at the right time but is unusable can reduce efficiency just as much as not returning any feedback at all. For example, developers may decide to ignore test results if they are delivered in a format that requires too much of their own investigation and filtering to find what they need to know. This increases the risk that they will be distracted by more defects later. It is also a waste of time and resources for SQA to run tests that benefit no one.

SQA also needs to understand how feedback cycles are related. For example, a software release delivery cycle may consist of the following separate cycles:

- Create a new release and confirm that it has been built successfully
- Test the new release and review the results
- Investigate reported failures and identify their causes
- Fix the failures and run regression tests

Each part of this cycle provides input to the next part. A build is delivered to test. Test results are delivered to developers. Developers act upon the results. A new build is retested. Teams need to understand not only how to complete their parts of the cycle, but also what they need from other teams and what other teams need from them to complete their parts.

In general, SQA organizations need to understand what they do right and wrong from the Development perspective, and Development needs to understand what they do right and wrong for SQA. For example, SQA could be delivering test results to Development at the right time, in the right format and with the right content. If Development is not able to review the results and fix failures as expected, SQA can't regression test in a timely manner and the cycle becomes inefficient.

2.1.3 Unique Needs

Developers have needs beyond efficiency and feedback. The needs could be related to the way the development team works by itself or the way it works with other development teams. The needs could be unique to the project or the current state of the project. The needs could also just be things that make the development team feel confident about their work and optimistic about the future success of their project. It might be difficult for developers to articulate these needs. It may be easier for SQA to suggest what they suspect might be a unique need to encourage developers to confirm or deny it. SQA should attempt to meet any unique developer needs.

2.1.4 Measurements

SQA needs to understand the measurements that are valuable to Development. These will likely include measurements related to efficiency and feedback, such as "On Time Delivery" and "First Time Right" measurements. Examples of these include tests passing, defect reports that contain all required information, and tests that are available to run when code is available to be tested. SQA should confirm whether there are specific targets for these measurements, such as verifying defect fixes within a certain amount of time of the fix being available. SQA should also confirm whether developers are interested in tracking trends, such as daily time spent waiting for test results throughout the project.

SQA and Development should also discuss whether legacy[1] measurements are still important and accurate. Some legacy measurements might be misleading with a new Development-Driven Testing strategy. For example, on legacy projects, test execution time may have been used to measure SQA efficiency. By this measure, SQA becomes less efficient if more time is spent during test execution collecting better failure information that then allows defects to be corrected quicker. SQA and Development may decide that completely new measurements are needed. Development and SQA should also keep in mind whether there are measurements that can prove that the new testing strategy is achieving better results than the legacy testing strategies.

2.2 Assessing SQA Capabilities

In addition to knowing the needs of Development, the SQA organization needs to know whether it has the capabilities to meet those needs. One way to assess this is to examine how the SQA and Development teams currently work together. The components of their working relationship can be thought of as:

- Interfaces
- Information
- Interactions

Once these are understood, it will be clear whether SQA can meet Development needs without changes.

[1] In this paper, the term legacy simply refers to anything that has been done before and is not a change for Development-Driven Testing.

2.2.1 Interfaces

Interfaces are the channels through which information is exchanged and the rules that govern their use. All communications require an interface. Interfaces include face-to-face discussions, e-mails, instant messages and phone calls. They also include tools and processes where information is produced, processed, stored and exchanged. Anything that requires the participation of both SQA and Development requires an interface.

Interfaces have to be used correctly for teams to work effectively. For example, the defect tracking system is an interface between SQA and Development. Incomplete defect reports can force developers to gather missing information on their own, making them less efficient. Using this interface correctly would help Developers maintain efficiency.

It should be clear to SQA which interfaces are needed to support Development requirements. SQA should have Development explicitly confirm them. SQA should not assume that interfaces that exist with other teams exist for the current team. Assumed interfaces that don't in fact exist will reduce efficiency and affect the timeliness and correctness of feedback.

2.2.2 Information

Information is the content exchanged through an interface and the format that content takes. Information can include code, tests, specifications, test results, builds, status, metrics, defect reports and schedules. Both the content and format are defined by the interface. Interfaces won't work effectively if the information does not conform to the interface requirements.

For example, tests are created using an interface. Requirements to be tested are identified, tests against those requirements are created, and those tests are reviewed to ensure that they are appropriate. A test development interface may define a process where requirements and tests are developed collaboratively between Development and SQA, and where test reviews are concurrent with test development. The interface may also define how those tests are documented so that they are efficient to write and review.

Information content and format should always be appropriate for the intended user and use. This will ensure that the information can be used efficiently. Deficiencies in testing information format and content will often be indicated by frequently repeated questions from developers. For example, developers may always ask whether a test failing on the current build passed the last time the test was run. Delays while waiting for answers can reduce developer efficiency.

2.2.3 Interactions

Interactions define when, how often and who is exchanging information using an interface. Interactions can include developers delivering a daily build to SQA, SQA running tests on a daily build, and SQA delivering daily build test results to developers. Feedback cycles always involve interactions.

Interactions are scheduled to occur when the information involved is most valuable. For example, developers benefit most when they receive test results immediately after they have made code changes. Waiting to find out if code changes worked is not efficient and delays development progress. Teams should also consider what could go wrong during an interaction so that there is adequate time to solve any problems that arise. For example, SQA could complete test execution as scheduled but leave no time for developers to fix any failures before the software release is due.

Teams should also ensure that any resources required to successfully complete an interaction will always be available, or agree on what would be an acceptable alternative. For example, if a specific hardware test fixture is not available for SQA to use, the teams could agree on an alternative fixture. The teams could also agree that the test should be delayed until the correct hardware is available. It is important for interactions to produce something of value, like credible test results, not simply take place.

Interactions can also affect and be affected by other interactions. SQA needs to understand how interactions are related. For example, SQA should schedule system testing to begin only after component testing has completed. This should be apparent to SQA since the system testing interaction can't being if the interface requires results information from the component tests, which isn't available until the component tests complete.

2.3 Optimizing the Testing Strategy

SQA and Development need to work well together to be efficient. Even teams that are working well together may need to become more efficient based on project constraints. SQA teams can optimize their interfaces, information and interactions with development teams to improve efficiency. Optimizations can include:

- Creating new interfaces or interactions
- Improving the information provided to interfaces
- Training staff to correctly use interfaces between Development and SQA
- Replacing ineffective interfaces, information and interactions

For any optimizations, SQA should confirm that the changes are having the intended effect. For example, SQA may find that developers are not processing test results quickly enough. To improve responsiveness, Development and SQA could decide to have SQA run tests less often, report results to developers less often or provide additional filtering of results so that only the most critical failures are reported to developers. (Development could also decide to increase their capacity to respond, but that is outside the scope of testing.) The teams should measure whether developer efficiency is improving.

Sometimes optimizations can have the opposite effect. For example, SQA may begin reporting test results to developers more frequently because SQA automated their manual tests. SQA may have expected this to improve Development efficiency but instead it made developers less efficient because of the additional time needed to process the more frequent rest results. Changing the frequency of test execution or results reporting could recover development efficiency.

Changing tools and processes can improve interfaces too, though careful thought has to be given to how those tools and processes will be deployed to the teams and how the teams will be trained to use them. Inadequate planning will likely reduce developer efficiency. For example, Development and SQA may determine that a new defect tracking system will improve developer efficiency. Putting the new tool in place without any training will likely reduce efficiency. Deploying the new tool in the middle of a critical period in the project will also likely reduce efficiency, even if the team has been properly trained. SQA needs to know not just what optimizations to make, but when to make them.

Tool changes can also optimize information. For example, if critical information is often missing from defect reports, it may be more effective to change the defect tracking system to require that information than it is to train users to always include it voluntarily. Tool changes can also optimize interactions, such as using an automated test system to execute tests whenever there are check-ins rather than relying on manually initiated test execution.

Interface optimizations can include improving SQA judgment about which interface to use when, such as when to make a phone call instead of submit a defect report to notify a developer about a new failure. This can have a surprisingly significant impact on how well the SQA and Development teams work together.

SQA and Development need to agree on the optimization strategy. While the SQA team may be able to implement some optimizations entirely within the SQA team, other changes may require the cooperation of the developers and program management. SQA needs to ensure that the project team understands what is involved in implementing the new testing strategy, how that strategy will be deployed and what benefits the team can expect.

3 Applying Development-Driven Testing

The following describes the software testing strategy for the development of a new print engine[2] component for a large print/copy/scan/fax system. This was a "clean sheet" design delivered over 5 years involving 20 software engineers, 200 mechanical engineers and 5 SQA Engineers collocated at one site. The print engine was developed iteratively, meaning:

- Enough code was written to bring up a new prototype revision
- Development learned from that prototype how to create the next revision
- Old code was discarded and new code was written to run the new prototype revision

SQA was engaged in the project following the initial iteration. Over six months, SQA and Development collaborated to produce the list of Development needs, the assessment of SQA capabilities and the new testing strategy optimized to meet Development needs.

3.1 Identifying Development Needs

There were several program constraints that influenced the needs of the development organization:

- Over 200 engineers were involved in the iterative creation of the hardware and software. The daily cost to the program for this many engineers was incredibly high. Even a single day of lost productivity would be an unacceptable expense and anything that significantly reduced the long-term efficiency of the team would put the entire program at risk.
- Iterative hardware development meant that prototypes would be hand-built and extremely expensive. Software developers and SQA would have few, if any, prototypes for software development and testing. Mechanical engineers would need to be able to use whatever prototypes they have for as long as possible.
- In addition to the print engine, the rest of the components making up the complete system would be new too. Integration of these components into a system would not be at the same site as print engine development. There would be at least an 8-hour delay between finding a problem at the integration site and engaging a print engine developer to investigate.

Based on these constraints, the following critical Development needs were identified:

- Immediate Feedback. Failures needed to be found immediately and fixes needed to be provided during the same day so that a stable baseline always existed for further development.
- Support for Multiple Simultaneous Builds. The program would need to extend the lives of prototypes as much as possible. It would be too costly to replace all prototypes whenever a new revision was released. Software would have to be written, maintained and tested for all prototype revisions in use. Development and SQA would have to create and ensure working software for multiple prototype revisions without regular access to hardware.
- Component Qualification. The print engine should never be the source of system integration problems and always needed to perform as expected. Any print engine problems found during system integration could never be addressed in a timely manner and would always create unacceptable delays and distractions for the print engine team.

[2] A print engine is the part of a printer that actually puts the image on the page.

3.2 Assessing SQA Capabilities

For each critical need, SQA Lead assessed the existing interfaces, information and interactions in use during legacy projects and determined whether they would be capable of meeting the needs of the print engine software developers.

3.2.1 Immediate Feedback

SQA investigated what generated feedback to developers, how often it occurred and whether it was sufficient to support the critical needs of Development.

- Test Development – Can tests be developed and maintained quickly enough to provide immediate feedback? When are the tests needed? Who is involved in creating the tests?
- Test Execution – Can tests be run frequently enough to provide immediate feedback? When do the tests need to be run? How often? Who needs to run the tests?
- Results Reporting – Can results be reported clearly enough so that they can be acted upon immediately? Who needs the results? When are the results needed?

3.2.1.1 Test Development

Current Status: SQA developed tests against traditional specifications. The specifications were written by developers and identified testable requirements. SQA developed test cases against the testable requirements and developers reviewed the tests to ensure that they were appropriate. Tests were documented to show traceability to requirements and written in a clear, concise manner to allow a wide range of reviewers to provide useful feedback. Specification changes were communicated to SQA through the defect tracking system.

New Requirements: Developers would not be writing any specifications. Software development would be driven by the results of iterative experiments rather than static requirements. Any specifications that were written would not affect print engine software development, would only be used by SQA to develop tests against, and would require constant maintenance. SQA would still need to demonstrate test coverage against requirements in a reviewable format.

Assessment: SQA would need to create a new way of developing tests without specifications written by developers and anew way of keeping those tests up to date despite constant code changes.

3.2.1.2 Test Execution

Current Status: Test execution cycles typically lasted from one week to two months depending on the scope. Testing cycles contained both fully automated (no human interaction required) and manual (human interaction required) testing, with tests run on prototype revisions most closely resembling the production units external customers would receive. The release tested during planned testing cycles could be a daily build, a weekly release or a release corresponding to a program milestone.

New Requirements: Testing the print engine would require more frequent testing cycles. Rather than one week to two months, testing cycles would need to complete in one day. Each testing cycle would need to be a complete regression test of all print engine functionality. SQA budget constraints would prevent hiring enough manual testers to execute a complete regression test every day. Budget constraints would also prevent purchasing and maintaining a sufficient number of prototypes to execute a complete daily regression test.

Assessment: SQA would need to create a new way of executing tests that did not require a large number of manual testers and prototypes and that could complete a full regression test daily in time for developers to make use of the results.

3.2.1.3 Results Reporting

Current Status: SQA reported results weekly to a cross-functional team at the program status meeting. For weekly testing cycles, the report would summarize the entire testing cycle. For longer testing cycles, the weekly report would show the progress being made against the test cycle plan. The report was compiled by hand and included data from the test management database as well as test results communicated through e-mail and spreadsheets. The finished report was e-mailed to the project team members.

New Requirements: Testing the print engine would require more timely reporting of results. Results would have to be delivered daily and at a time when developers could act upon them effectively. The results would need to contain information targeted to developers rather than a cross-functional program team and would need to be in a format that assisted developers in identifying and isolating regression failures. All the information needed to investigate a failure would need to be included with the results.

Assessment: SQA would need to create a new way of reporting results that supported daily regression testing and defect correction.

3.2.2 Support for Multiple Simultaneous Builds

Current Status: SQA always tested one software build at a time. The software build was for the most recent, customer-intent prototype. SQA would frequently have to upgrade prototypes during the course of a project so that tests were always run on customer-intent hardware. Testing qualified the hardware and software for external customer use.

New Requirements: Print engine testing would need to focus on enabling hardware and software developers to sustain their development efforts. Multiple, simultaneous builds would have to be tested to allow development to continue on any available prototypes. All requirements for test execution, including running a complete regression test and reporting results daily, would have to be met for each software build for each prototype revision in use.

Assessment: SQA would need to change the way it executed tests and reported results so that multiple, simultaneous builds could be tested concurrently.

3.2.3 Component Qualification

Current Status: SQA only tested integrated systems. No components were tested separately. Software releases contained all components integrated into a system and system tests were run on customer-intent hardware prototypes. Human testers interacted with the devices in the same way that end users would.

New Requirements: Testing the print engine would require testing the component without integrating it with the rest of the system. Tests would have to use some other mechanism of interacting with the print engine software than with the physical hardware interfaces (front panel buttons, doors, trays, etc.…). Sufficient hardware would not be available to SQA to run the tests on prototypes.

Assessment: SQA would need to create tests to qualify the print engine component in isolation of the rest of the system and the print engine tests would need to run without prototype hardware.

3.3 Optimizing the Testing Strategy

The SQA team needed to create a new strategy for testing print engines. It needed to adapt test development, execution and results reporting practices to provide software developers with immediate feedback. SQA needed to create new methods to test multiple, simultaneous builds and test software components in isolation of the rest of the system. And it needed to be able to execute this strategy while meeting budget constraints for the program.

The new strategy involved the creation of:

- A simulation environment for testing print engine components without hardware.
- Fully automated tests requiring no human interaction for execution or results reporting.
- A scalable test infrastructure that supported testing multiple, simultaneous builds.

SQA also changed the interfaces, information and interactions involved with developing and executing tests, reporting results and investigating failures.

3.3.1 Test Development

To address test development, SQA Engineers collaborated with developers to create test documentation that could act as specifications by example. This eliminated the need for separate specifications and improved developer efficiency. Understanding the requirements and tests before coding also prevented design and implementation problems that would have reduced developer efficiency. Constant face-to-face conversations between developers and SQA Engineers ensured that the tests were always up to date and correct. All tests were developed in parallel to the print engine code so that tests were available to run as soon as the code was ready to be tested.

3.3.2 Test Execution

Several changes were needed to enable SQA to run a complete regression test of the print engine component without people or prototypes.

- SQA collaborated with developers to create a simulation environment using only a print engine controller board and a power supply. All interfaces to external hardware were simulated in the print engine code. With the code running on the actual controller, testing would ensure all real time requirements were met. The cost of the print engine simulator was $1/100^{th}$ that of a full prototype, allowing SQA to have an adequate supply for parallel test execution.
- SQA and developers created a new programming language to simulate communications with other system components. This allowed the print engine component to be tested in isolation. The programming language implemented the same communications protocol used by all the components in the system.
- SQA created automated tests. No manual tests were written. The automated tests could be run without modification on a simulator or a hardware prototype. While tests would run without any human interaction on simulators, they would prompt the test operator to interact with the device when run on a hardware prototype. Developers ensured that the simulator supported all actions needed to execute tests and verify results.
- SQA created a scalable test environment that allowed test execution to be distributed across multiple simulators. Automated tests were structured to support parallel test execution, with no single test suite taking more than 1 hour to complete. This allowed the number of tests to grow, yet still have test execution complete in the same amount of time, by simply running the tests on more simulators. Testing multiple builds only required adding more simulators.
- The SQA automated test environment allowed developers to select and run automated tests. Automated tests initiated by developers would run exactly as they ran during planned SQA testing and would contain all the same results, links and logs.

3.3.3 Results Reporting

Many changes were required to ensure that results were reported at the right time, that the results were actionable and that developers acted upon them.

- SQA designed the automated test environment to monitor the daily build status, upgrade simulators with the latest build, and execute tests. The test environment captured and stored all logs generated by the simulators, test environment and automated tests, and associate them with the individual test results.
- The test environment automatically packaged up all results, links to tests and log files, and built an e-mail message. The results e-mail was sent to all developers. Because no human interaction was required, the end-to-end process could run as soon as the daily build was available, typically at 1:00am, and results would be waiting in the inboxes of developers when they started their day.
- The format of the test results e-mail allowed developers to see the most important information without scrolling. Developers could quickly tell whether any further reading was required for that daily build.
- Test results were organized by software subsystem and named using the same terms used by developers. This made it intuitive to find test results of interest.
- Test results were listed with simple tests first. Test results for complex tests were built upon the known results of the simple tests. This made defect isolation more efficient.
- The report highlighted differences from the previous testing cycle (yesterday's build) so the impact of changes would be clear immediately. Differences included tests that were now failing as well as now passing.
- On the occasion when test failures were not immediately addressed, the failures were annotated in the report with references to defect reports. Annotations also included the number of days the failure had persisted in the code. This allowed developers to quickly identify new failures.
- SQA created a series of automated tests to determine quickly which features were supported in each new build. The test environment only executed the tests for which all the required features were supported. This eliminated obvious failures from the report that would otherwise be a distraction to developers.
- Software development managers ensured that failures were addressed as quickly as they were reported. Managers reviewed daily test results and contacted their developers directly to ensure that failures would be fixed and retested that morning. Failures where the fixes that could not be delivered immediately were addressed by backing out the changes and rebuilding the code. The SQA team provided the same responsiveness to test and test environment failures.
- SQA investigated, isolated and recommended fixes for print engine failures. SQA Engineers reviewed the daily test results against the check-ins from the previous day and contacted the developers to discuss the possible causes of failures. This helped to pinpoint defects more quickly and became a critical service for the development team.

3.3.4 Support for Multiple Simultaneous Builds

Development agreed to create and maintain builds for the current prototype and the previous two revisions. SQA tested all three builds daily. Automated tests and the test environment code branched along with the print engine code for each new prototype revision so that the right tests would be run in the right environment for each build. Separate results reports were e-mailed for each build tested.

3.3.5 Component Qualification

SQA verified all print engine component requirements by executing a complete automated regression test on print engine simulators. SQA supplemented the simulator tests to ensure that the print engine component would work correctly in a fully integrated hardware system. This included:

- Verifying the system communications protocol requirements. A complete automated test was run as part of the daily regression test to verify that the print engine component met all communications protocol requirements.
- Verifying that the print engine ran correctly when integrated into the complete system. Once a week, simulator testing was supplemented with testing on a real device. The 4-hour test covered the highest priority features and could be run by one person on one prototype.
- Running a complete regression test on a hardware prototype prior to every significant program milestone. The test, which took 80 hours to run on simulators, took one human tester 6 weeks to run on a prototype. SQA planned for these testing cycles to complete a month before each program milestone. This provided developers with enough time to fix any print engine problems before the component had to be released for system integration.

3.4 Impact

The print engine went through 8 prototype iterations. The final version of the software contained 450,000 lines of code. When the product was released, it was the most reliable print engine in the company's history. The testing strategy was credited with reducing the duration of prototype iterations and overall product development. The print engine was never the cause of integration failures during 5 years of SQA engagement and automated testing. Automated testing and hardware simulation saved the company tens of millions of dollars over manual testing on full prototypes. By eliminating most printing of physical pages, hardware simulation alone reduced paper and ink expenses enough to pay for the entire SQA effort.

Executive management enlisted other teams to investigate how the methods could be applied to their programs throughout the company. The automated test environment was replicated at other development sites and enhanced to test integrated systems as well as print engines. SQA also applied the Development-Driven Testing methods to other internal customers, including technical publications, manufacturing, customer support, field service and program management.

4 Conclusion

SQA organizations need to see software developers as customers for their testing services and ensure that the services provided absolutely meet their customers' needs. This requires understanding how the development team works, what is important to them and how testing fits into the process. SQA organizations need to reassess with each new project and each new development team whether their current testing practices are adequate. Crafting new testing strategies becomes a collaborative effort between SQA and Development.

Development-Driven Testing provides a framework for discovering the needs of developers, analyzing the interfaces, information and interactions between Development and SQA and assessing whether current SQA practices are capable of meeting Development needs. The DDT framework can be applied to other internal customers in addition to software developers.

SQA organizations can do more than validate products for end users. They can use testing to improve software development efficiency, reduce software development costs and improve product quality. SQA can have a significant impact on how products are developed and how successful they will be.

Acknowledgements

I would like to thank Mike Herbert, Jeff Lowe, Lynd Wieman, and Jeremy Smith for their contributions to this process, and to Tektronix for teaching me about internal customers many years ago. Special thanks to my talented reviewers for helping me turn this into something that people will want to read. Extra special thanks to David Butt for suggesting the name Development-Driven Testing. I like it!

Recommended Reading

While the name Development-Driven Testing may be new, the idea is not. It is focused on providing excellent SQA services to internal software development customers to improve software development efficiency. DDT is best understood as an outgrowth of many generally accepted practices from the past few decades.

Beck, Kent 2000. *Extreme Programming Explained*. Addison-Wesley.

Coplien, James and Harrison, Neil 2005. *Organizational Patterns of Agile Software Development*. Pearson Prentice Hall.

Gilb, Tom 1988. *Principles of Software Engineering Management*. Addison-Wesley.

Grady, Robert 1992. *Practical Software Metrics for Project Management and Process Improvement*. Prentice Hall.

Poppendieck, Mary and Poppendieck, Tom 2003. *Lean Software Development*. Addison-Wesley.

Spradley, James 1980. *Participant Observation*. Holt Rinehart Winston.

Weinberg, Gerald 1986. *Becoming a Technical Leader*. Dorset House.

Weinberg, Gerald 1989. *Exploring Requirements: Quality Before Design*. Dorset House.

Process Dynamics, Variations and Controls in Software Engineering

Shivanand Parappa
shivanand.parappa@cognizant.com

Abstract

Business operations are always under the influence of variations due to change in customer needs, market demands, internal processes and technology upgrades. These changes have direct impact on the performance of IT processes cascading down to the quality of products/services. In order to mitigate the adverse effects due to the variations of the aforementioned, organizations need to understand the concepts of software process dynamics and process controls.

Variations in IT processes are due to the factors affecting the process equilibrium. The enforced process controls are often not sensitive to these factors. This paper concentrates on the theory of software process dynamics and process variation including the importance of process controls to ensure process equilibrium.

This paper focuses on how the understanding of process variation and controls are useful in sustaining and improving process capability. This paper discusses the degree of association between Process and Quality through a Process Quality Relationship (PQR) Matrix that helps in managing the processes and sustaining the product/service quality.

This paper expands the possible ways and means to define and develop organizational process assets from a Total Quality Management perspective. This paper details an approach named SDICC (Strategy, Design, Implementation, Control and Continuous improvement) that provides guidance towards designing the processes and sustaining the improvements.

Biography

Shivanand Parappa is a Senior Consultant at Cognizant Technology Solution, currently working with Process and Quality Consultancy team. He has over 16 years of experience in Manufacturing / Software Quality Management and worked in various positions ranging from Industrial Engineer to Quality Manager. In his previous assignments, he played a key role in implementation of management system models like CMMI, ISO-9001, ISO-14001, OHSAS-18001, International Quality Rating Systems (IQRS). He has extensively worked on driving process improvement projects through Six Sigma and Lean methodologies to deliver tangible results.

Prior to Cognizant he worked with Patni Computer Systems, Tata Consultancy Services, Sterlite Industries and Grasim Industries. Shivanand holds a Bachelors degree in Industrial Engineering and a Masters degree in Production Engineering. He is a certified professional with Six Sigma Black Belt (DMAIC and DFSS), PMP and PRINCE2 Practitioner.

1 Introduction

Business operations are always under the influence of variations due to change in customer needs, market demands, internal processes and technology upgrades. These changes have direct impact on the performance of IT processes cascading down to the quality of products/services. In order to mitigate the adverse effects due to the variations of the aforementioned, organizations need to understand the concepts of software process dynamics and process controls.

Process helps an organization's workforce to meet business objectives by helping them to work smarter, not harder, and with improved consistency *[CMMI-DEV v1.3 2010]*. Processes are critical entities in an organization and its operational success is tied with the effective performance of the processes. One of the never ending challenges for an organization is to act on customer requirements and translate them into product or service with agreed quality attributes at low cost. The actions involved in conversion of the requirements to products/services are enabled and supported by well defined processes coupled with adequate resources. Processes can be grouped and classified into: core; enabling; governance; or support processes. This classification is only to bring the distinctions in the organization process asset instead of indicating its priority or ranking. Each of the organizational processes has equal importance with respect to achieving the objectives and goals of the organization.

"A process is a set of activities that are interrelated or that interact with one another" [ISO-9000:2005]. During the execution of process, the supplied inputs are converted into desirable output which may be intermediate or final products/services depending on the complexity of the process. Monitoring and control actions are imposed on the process to address deficiencies found in the process output. The variation in process performance *"voice of the process" [Howard Gitlow 2009]* is measured in terms of efficiency (output quantity against the target) and effectiveness (output quality against expected). Necessary actions are executed on activities within the process and supplied inputs to minimize or eliminate the variations. On the other hand, changing customer needs *"voice of the customer" [Kai Yang 2007]* are captured and incorporated accordingly into supplied inputs to meet customer expectations.

The following block diagram (Figure1) depicts the process model with closed loop feedback mechanism along with the changing customer needs (voice of the customer) and variation in process (voice of the process).

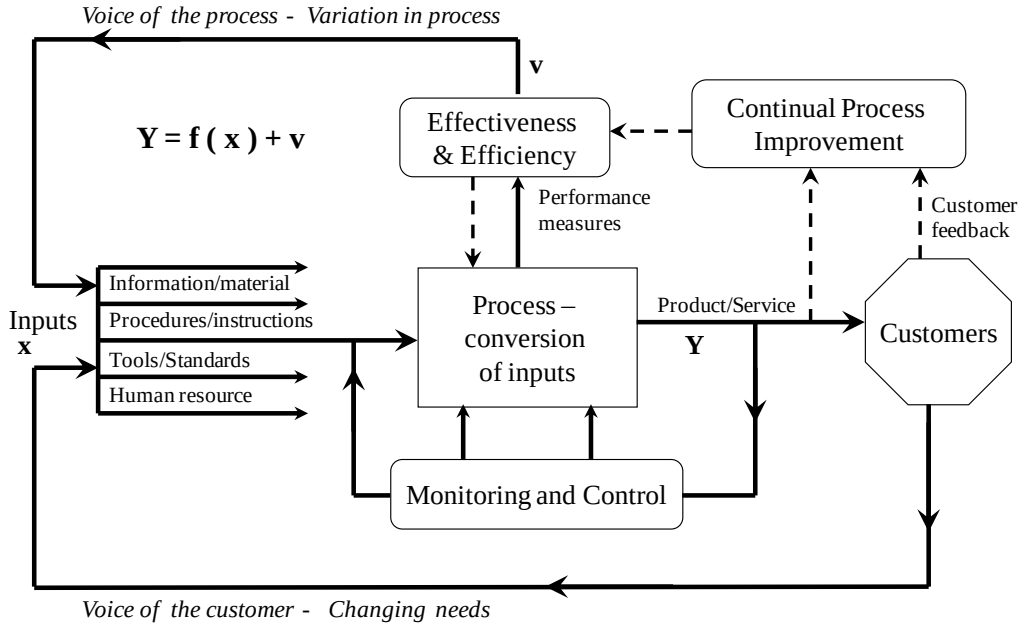


Figure 1 Closed loop process mechanism

From experience, it is observed that product/service quality is directly proportional to the process quality with some amount of inherent process variations due to common causes. *"The quality of a system or product is highly influenced by the quality of the process used to develop and maintain it" [CMMI-DEV v1.3 2010]*. From Six Sigma perspective, this relationship generally expressed through following concept *[Craig, Bruce and Terry 2006]:*

$$\mathbf{Y = f(x) + v}$$

Product/service quality, $\mathbf{Y}$, is function of process quality $\mathbf{x}$ (supplied inputs, activities and controls) with a certain degree of inherent process variations $\mathbf{v}$. For example, quality (defect free software) $\mathbf{Y}$, depends on the parameters $\mathbf{x}$ (requirement clarity, design approach, coding process, test cases and review methods) with some inherent process variations $\mathbf{v}$ (people skill level in coding and reviews).

2 Process dynamics

Among the various engineering domains, software engineering is very peculiar due to its dynamic nature of operations and variations in process implementation. One of the reasons for varied process dynamics is that software engineering processes are heavily dependent on rapid changing technologies and involvement of human beings rather than machines. From a physics point of view, dynamics is a branch of mechanics dealing with motion of material bodies under the action of given forces. This definition holds good for software engineering also where flow of information and processes are under the influence of forces like change in business operations, customer needs, technology, etc. Software process dynamics refers to the factors affecting software processes over a time period. It is also an approach to understanding the behaviour of processes based on the impacts of external and internal factors.

The dynamics of software processes has been explored by process practitioners to understand its mechanism. Due to competitive market and increased economic pressure, the ability to understand about dynamics and complex software development processes has become increasingly valuable for decision making. Process instability is due to various factors such as:

- Combined effects of schedule targets
- Communication overhead
- Changing business conditions
- Requirements volatility
- People experience
- Work methods such as reviews and quality assurance activities
- Task underestimation and organizational shifts.

Because of these factors, the adverse effects of process dynamics for individual sub-processes aggregate to the parent process level causing a large impact on intermediate/final deliverables and products/services.

To understand overall process dynamics, it is required to decompose processes into sub-processes and identify the associated forces at each sub-process level. The relationship between parent and sub-processes can be expressed in terms of a theoretical equation as shown below.

$$\sum_{\mathbf{i=1}}^{\mathbf{n}} \mathbf{Pdy_i} = \mathbf{pdy_1} + \mathbf{pdy_2} + \mathbf{pdy_3} + \mathbf{...} + \mathbf{pdy_n}$$

Where, $\sum \mathbf{Pdy}$ is overall process dynamics,

$\mathbf{pdy_1} + \mathbf{pdy_2} + \mathbf{pdy_3} + \mathbf{...} + \mathbf{pdy_n}$ are process dynamics of each sub-process and

$\mathbf{n}$ is number of sub-processes

Process dynamics relates to variation in the process performance measurement/metric (like productivity, effort and cost) of a process. Process metrics could be critical to quality or process parameters through which a product or service accepted by customers. In-depth understanding of behavior of sub-processes helps in establishing process performance (prediction) models which can be used to calculate the expected performance level of a parent process based on current performance level of the sub-process. For example, performance of peer review process (measured as overall defect density) is associated with performance of sub-processes like design review and code review and also technical experience of the people involved.

From the above equation, we can see that overall process dynamics is the sum of cumulative variations of individual process or sub-processes. Variations in processes are due to factors affecting the process equilibrium which are not sensitive to the currently enforced process controls. Most of the time, it is quite possible that controls are not adequate to minimize the effect of the factors causing process turbulence and not tested for their significance before deployment. Requiring processes to bear the impact of process variations without producing an adverse effect on performance is a key requirement for process definition. In order to establish rigorous controls over the defined processes, a detailed study and in-depth investigation of the causes for process variation are required.

The following schematic diagram (Figure 2) represents a simple process dynamics model for software development and its interrelated sub-process components (shown in boxes).

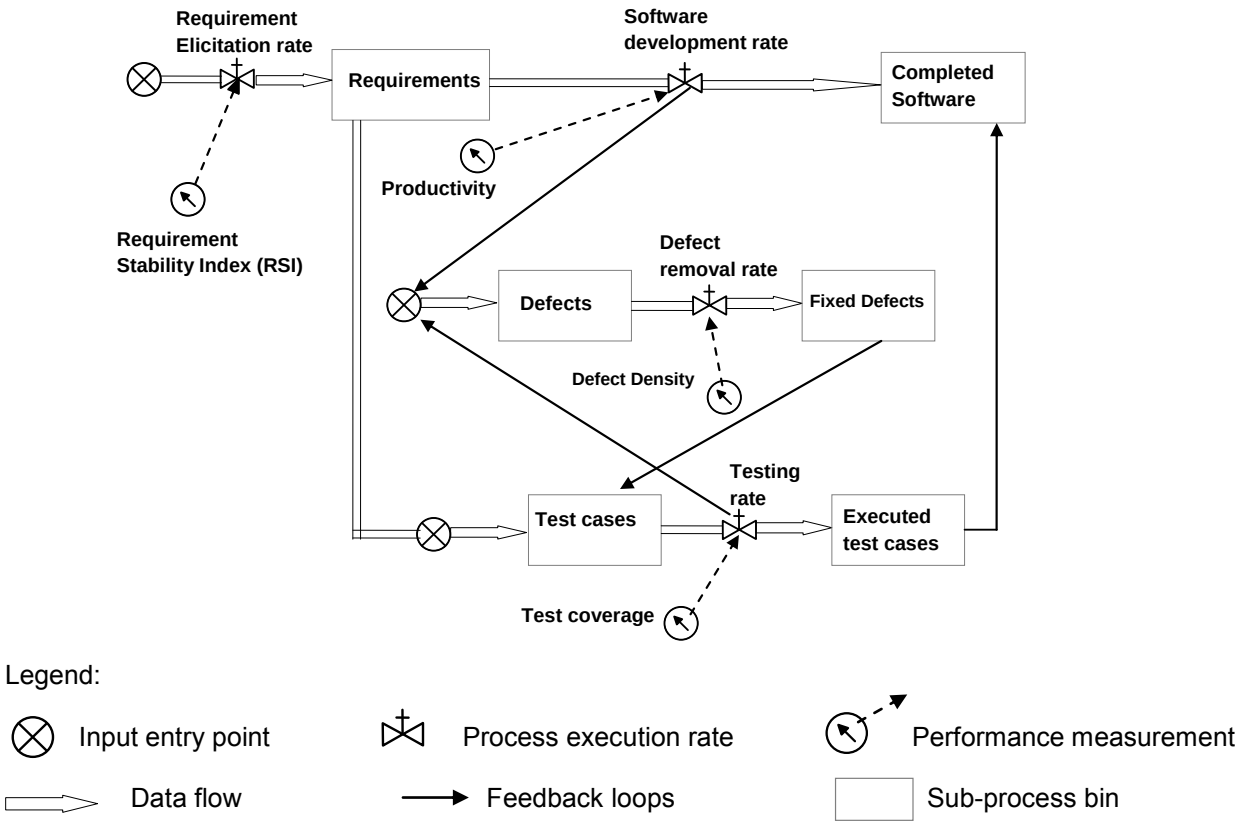


Figure 2 A Simple software process dynamics model

In this model, software is developed with an objective to fulfill the given requirements at a certain development rate which is measured in terms of productivity. The requirements are transformed into completed software at the software development rate. The completed software, which is the outcome of the software development process, depends on the performance of the sub-processes such as defect identification and removal (for example, number of defects identified and fixed per day) and test cases

execution (for example, test cases executed per person day). Each of the sub-processes consumes inputs and produces the necessary outputs to achieve the intent of the main process output (completed software).

During the course of software development, the outputs of one process become inputs to the other. Efficiencies of individual processes decide the overall efficiency of the main processes. As time progresses, requirements are processed at the software development rate and requirements become developed software, so level (quantity) of requirements decline as level of completed software development activities rises. The software development rate is constrained by several factors: the defect removal rate, testing rate and nominal productivity of people. In order to keep the completed software delivery schedule unchanged, there must be a balanced flow rate between sub-processes in the system.

From the system dynamics perspective, to ensure process equilibrium, levels in the identified bins (requirements, completed software, defects, fixed defects, test cases and executed test cases) must increase or decrease proportionately according to the flow rate of each of the sub-processes. For example, the level of test cases and executed test cases boxes vary according to the testing rate. Additionally, the test cases box level changes with respect to the requirements elicitation rate. Subsequently, the expected level in a completed software box is achieved and maintained along with the flow rate of other sub-processes. Organizations need to study the causes for variations in flow rate and uneven levels to ensure smooth and effective process flow and performance.

3 Process capability versus Process controls

Process stability is ensured by bringing processes under statistical control through execution of the appropriate control actions against special causes. These actions may be for process deviations that have already occurred (corrective actions) or could be for potential deviations predicted based on process behaviour trend from control charts (preventive actions). On the other hand, process capability is a statistical indicator of how well a process is functioning. The increment in process capability indices over a period of time depicts the effectiveness of control taken against common causes responsible for inherent variations. In fact, control actions for common causes are nothing but process improvement actions taken to reduce inherent variations. In statistics, the degree of process variation is measured in terms of standard deviation and capability of a process expressed in terms of process capability indices **Cp** and **Cpk** (**Cp** - assumed process mean is centred and **Cpk** - assumed process mean is not centred) *[Thomas and Paul 2009]*:

$$\mathbf{Cp = (USL - LSL)/\ 6 \times standard\ deviation}$$

$$\mathbf{Cpk = Minimum\ of} \left\{ \mathbf{(USL - Mean)/3 \times standard\ deviation,\ \ (Mean - LSL)/3 \times standard\ deviation} \right\}$$

Where, **USL** - Upper Specification Limit of process metric and
LSL - Lower Specification Limit of process metric

Once a process achieves an incremental change in its process capability index, a new set of control limits with a narrower bandwidth are established and future process performance is measured against these limits. Subsequently, the process is monitored against new control limits. With the revised control limits, still a process will have some common causes with a reduced adverse impact on process outputs. Since common causes can't be eliminated completely, variations due to these causes always present in any processes. In order to sustain process stability and improvement results, organizations need to focus on designing and establishing rigorous control actions.

Process improvement initiatives must continue until process quality reaches Six Sigma level *"3.4 Defects Per Million Opportunities" [Basem and Adnan 2010]* or even beyond depending upon the criticality of the

process. The following diagram (Figure 3) depicts the association between incremental process capability, process improvement and degree of controls.

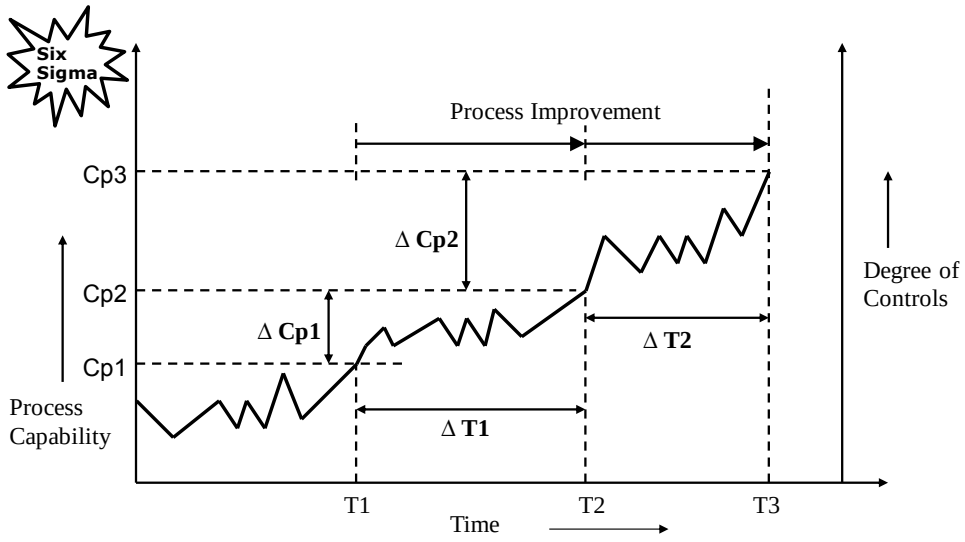


Figure 3 Process: Capability, Improvement and Controls

The relationship between process capability, process control and process improvement can be represented in terms of a time dependent linear function as shown below (for the purpose of clarity only process capability index **Cp** is used, however, **Cp** can be replaced with the other capability index **Cpk**) :

Δ Cp = f (process control × process improvement) × Δ T + inherent process variation

Where,

Δ Cp = Change in process capability over a time period,
(for example, **Δ Cp1** = Cp2 value at T2 − Cp1 value at T1)

Δ T = Change in time period (for example, **ΔT1** = T2 − T1)

Process control: actions to ensure process performance/output between established limits (for example, reviews/checks will control the defect density to remain in established limits).

Process improvement: actions to improve the current level of process performance/output (for example, enhancing the effectiveness in reviews/checks to establish stringent variation limits for defect density).

Inherent process variation: unavoidable process variation which cannot be eliminated or reduced further (for example, variation in competency and skills of the reviewer).

Referring to figure 3 and the above expression, the increment in process capability is a time dependent function of process control and process improvement with some amount of unavoidable process variation. The increment in process capability can be achieved over a period of time only when implemented process improvement actions are coupled with relevant and intensive controls. Process capability is affected by failure of an improvement action to address the causes of variations and/or imposing inadequate process controls. Some amount of inherent process variation is always bound to happen which is not economical and cannot be eliminated or reduced further.

For example, the current (at T1) process capability index (Cp1) for productivity of test case preparation process is calculated as 0.67 (a capability index of 2 is ideal value for a Six Sigma level process). Though

the process is stable (statistically under control), a low capability index indicates that the process is not capable enough to produce the outputs within the given specification limits. The process is analyzed and improvements actions are implemented with appropriate controls to improve the process. After four months (at $T2$) the capability index ($Cp2$) is calculated as 1.4. The change in capability index Δ **Cp1** (from 0.67 to 1.4) is a function of actions taken for process improvement (e.g., modifying the current test case preparation method) and deployed process control actions (e.g., use of templates) over a period of time $\Delta \mathbf{T}$ (4 months).

The above linear equation is valid as long as the processes are well programmed and futuristic (predictable) in nature. In other words, a process must be stable (under statistical control) and capable to meet process goals along with good control mechanisms in place to mitigate unfavourable conditions.

3.1 Characterizing the process variation:

Characterization of process variation is essential in mitigating the effect of variations on process performance and quality of process outputs. The design and establishment of process controls for variations are driven by the behaviour of the variations and their attributes and characters. Processes behave differently in different circumstances due to various factors and causes. Analysis of process variation patterns and process behaviour provide critical information for initiating process improvement programs and to establish appropriate process control. With the aid of appropriate statistical tools, the characterization of process variation can be done and inferences drawn from the analysis reveals the direction for further investigation to arrest the causes of variation.

3.2 Defining the root causes of variation:

Based on the preliminary inference report and characteristics of variation, the next step is mapping of process variation to the probable causes responsible for the variations. During this effort, care must be taken in identification of causes to ensure that only the relevant causes are mapped to the variations with sufficient analysis and evidences. As the identified probable causes are further evaluated to define the root cause and all proposed actions are based on the root causes, it is very important and necessary to use correct tools and techniques for causal analysis. Many different processes, tools, and techniques (e.g., Failure Mode Effects Analysis, Ishikawa diagrams, Pareto charts) are available for defect/variation causal analysis *[Thomas and Paul 2009]*. All of them have proven to be successful in various situations. While application of these techniques helps gain insight into the sources of problems, a checklist implementation of the techniques alone is not sufficient to ensure accurate identification and effective resolution of “deep” problems. After discovering a set of probable causes for the variation, with the aid of statistical and analytical tools, the exact root causes are identified and evaluated for their significance.

3.3 Controls and improvements:

Actions to eliminate the identified root causes of process variation are proposed in terms of controlling actions and improvement actions. For example, introducing a comprehensive test case review checklist to eliminate the cause ‘ineffective review’ leads to higher defect density in completed software. Controlling actions are nothing but the corrective actions taken to remove the root causes to avoid the recurrence of variations due to the same causes. Improvement actions are actions taken to improve process performance and capability by eliminating the occurrence of relevant potential causes for the variations.

An example of where improvement actions could be used is in the monitoring of software development productivity through a process control chart, in which actions can be taken to avoid the dropping of productivity value below some lower control limit and narrowing width of control limits by reducing the variation. The proposed actions should be evaluated for their significance with respect to the identified root causes through piloting and significance (hypothesis) tests. On deploying both the controlling and improvement actions, there must be a positive improvement trend in process performance and capability.

3.4 Actions for sustaining improvements:

The job of sustaining process improvement results is really a big task as most of the improvements last for a short time due to lack of focus on long term sustaining actions. Planned process improvement actions must address the root causes and appropriate measures for sustenance should be planned and inbuilt as part of improvement actions. A well established measurement and analysis system helps in monitoring and controlling the process performance and also ensures the sustaining of improvement in long run. The sustaining actions could include actions such as imparting training to process performers and owners; periodic review of performance results; adequate monitoring of process controls.

4 SDICC Approach for Process Management

In this section, a structured approach for process management and its components is discussed in line with the expectations for effective process management. This approach consists of five components (abbreviated as SDICC): Strategy; Design; Implementation; Control; and Continuous Improvement. The SDICC process management approach (See Figure 4) provides detailed insight on the requirements for developing a robust process definition and modelling strategy. Each component has its unique contribution and effect on the end-to-end process definition approach.

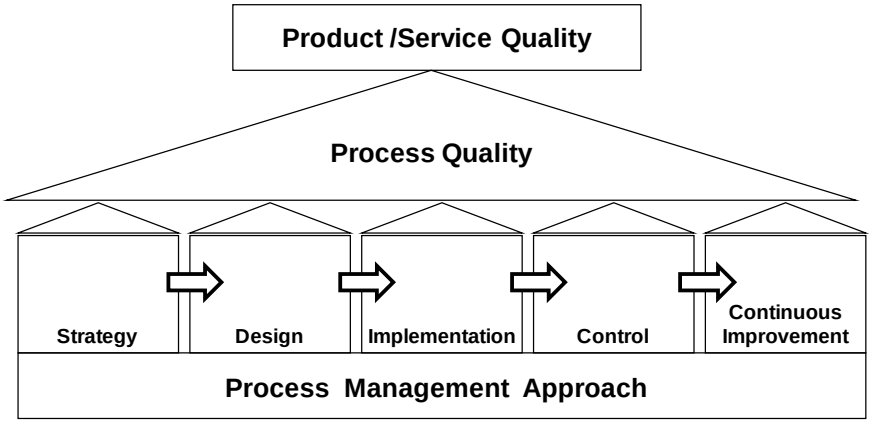


Figure 4 SDICC Approach to Process Management

The above Process management approach provides a structured approach in formulation of processes to ensure consistent product/Service quality. The results of activities within each of the components form inputs to next component. All the components collectively contribute in ensuring the process quality and in achieving the process objectives thereby producing quality product/service. The following paragraphs outline the need and importance of each component in determining the quality of processes and products.

4.1 Strategy

Process strategy is the pattern of decision making in identification, selection and managing of processes so that they will achieve their competitive priorities and build business value through process improvement. The initial information such as customer requirements, product features and organization initiatives should be considered as inputs when formulating process strategy. Process strategy must match with business needs, culture and specific issues of the context. During the course of process strategy, organizations need to consider the quality improvement objectives, current process infrastructure, tools, technology and competent resources. Strategic Analysis Tools like SWOT (Strength, Weakness, Opportunities and Threats), PEST (Political, Economic, Social and Technological) and other decision making tools *[Cornelius 2010]* provide a structured approach in the formulation of process strategy.

4.2 Design

Based on the result of process strategy, prioritized and selected processes are designed to meet process objectives and thereby enable organizations to achieve business and customer needs. The use of a quality function deployment (QFD) tool *[Thomas and Paul 2009]* helps in mapping between specific requirements (what) and methods to achieve them (how). The analysis of possible design failures and their effects is done through the Process Failure Mode and Effect Analysis (PFMEA) tool to ensure robust process design *[Thomas and Paul 2009]*. Practically, the process design involves the identification of process boundaries and dependencies, process and work flow definition and structured documentation. Process definition includes components like scope, purpose, entry criteria, inputs, outputs, tasks involved, roles and responsibilities, monitoring and controlling criteria. Process Definition lists what happens between the start and end points and includes all the activities that transform an input into an output. Quality planning and quality control aspects are built in during process design and definition.

4.3 Implementation

The trigger for process implementation starts once process definition completes. Initial process implementation is piloted on selected areas to ensure adequacy of the process design to enable subsequent process refinement before implementing across the organization. Prior to process implementation, all stakeholders of the process need to be trained on the processes that are planned to be rolled out. A process governance team (which governs the implementation activities) needs to be formed along with an appropriate communication/escalation structure to ensure effective implementation. Process owners and performers should be aware of the purpose, objectives and outcomes of the process along with the relevant measurements to be captured during implementation. All together, process implementation consists of communication, process training and mentoring, tool training, support mechanisms, metrics and infrastructure for continuous improvement in addition to the process definition itself. All associated artefacts must be generated in a timely manner and maintained according to defined guidelines and checklists. While processes are being implemented, the identified process measurements must be captured as per the defined frequency and metrics report generated to track process performance against the targets. Effective process implementation is ensured through use of tools like checklists, guidelines and metrics.

4.4 Control

The primary objective of process control is to track process performance against expected results and maintain steady performance. Appropriate corrective and preventive action needs to be taken in case of process deviation to bring back the process into its normality. The adequacy and degree of controls should be derived considering the stability and capability of the process. A measurements and metrics report should be generated for timely execution of the control actions to achieve the process objectives. Appropriate process control actions identified based on the degree of process deviation, process behaviour and complexity of the problem. Use of typical data analysis techniques/tools like Histograms, Pareto Charts, Scatter diagrams, Cause and effect diagrams and Statistical process control *[Thomas and Paul 2009]* charts help in identifying and selection of required process control actions. The usage and applicability of particular tool depends on the nature of the problem.

4.5 Continuous Improvement

The analysis results from process control provide opportunities for process improvement initiatives. Process control actions can be analyzed further to identify the causes of chronic problems and variations. Improvement opportunities identified with an objective to eliminate the causes or to reduce the effect of causes depending upon nature of the problem. The success in achievement of process improvement objectives lies in adapting and following an appropriate process improvement approach. Continuous process improvements could range from small improvements like Kaizen *[Besterfield 2011]* to break through improvements like Six Sigma. Organizations can choose a suitable process improvement model/approach like Juran Quality Improvement (JQI) *[Mukherjee 2006]*, Six Sigma *[Thomas and Paul*

2009], Lean Six Sigma *[Michael 2002]*, TQM *[Besterfield 2011]*, CMMI *[CMMI-DEV v1.3 2010]*, etc. based on the need and nature of the change or problem.

5 Process Quality Relationship (PQR) Matrix

Quality Management for macro processes is carried out by use of the Juran Trilogy *[Besterfield 2011]*, which basically consists of three stages: quality planning, quality control and quality improvement. These three stages, taken as a whole, form the basis for the entire quality management effort. This approach has been accepted widely and is used in all types of industries (example: Automobile, IT, Banking, Healthcare etc). The Juran Trilogy is essentially concerned with designing products and processes that align with customer needs. In this trilogy, quality is an integral part of the planning, control and improvement activities with a focus on reducing cost of poor quality. During the course of managing the process quality, organizations should consider the process management aspects (discussed in section 4) in developing and framing adequate processes. Though quality and process management activities overlap each other to a large extent while achieving the organizational objectives, the approaches for quality and process management vary with respect to the process execution perspective. For example, the quality improvement component of the quality trilogy provides necessary inputs like quality objectives, customer and market requirements to the process strategy formulation activity. Both process and quality management are an integral part of an organization's operations in achieving and sustaining the product/service quality.

5.1 PQR Matrix interpretation

The following Process-quality relationship matrix (Figure 5) shows the degree of association between process and quality components.

Process Management Components →

Juran Quality Trilogy ↓	Strategy	Design	Impleme-ntation	Control	Continuous improvement
Quality Planning	3	5	3	1	5
Quality Control	1	5	3	5	3
Quality Improvement	5	3	1	3	5

1 Weak 3 Moderate 5 Strong

The numbers represent the degree of association between quality and process components

Figure 5 PQR matrix

The five process management components are indicated along the columns and the Juran Quality trilogy is indicated along the rows. The degree of association between process and quality components has been quantified in terms of Weak, Moderate and Strong (Weak-1, Moderate-3 and Strong-5) to depict how both the process and quality blend together while process modelling. The quantification of relationship is based on the results of impact analysis of each component over the other during process design and execution while achieving the targeted quality.

The PQR matrix reveals the fact that, in totality the components of the quality trilogy across the process management components have equal importance though they have varied one-to-one degrees of association. For example, the relationship scores along the rows for Quality Planning (3+5+3+1+5),

Quality Control (1+5+3+5+3) and Quality Improvement (5+3+1+3+5) all total 17. This depicts the need of the attention of process practitioners for considering and ingraining the quality aspects while defining and managing the processes to achieve the set process objectives and quality outputs. The individual relationship value can vary depending on the scale used (current scale is: Weak-1, Moderate-3 and Strong-5) but the relationship attributes (Weak, Moderate and Strong) remains same. With a change in scale, the total score across the rows (current total is 17) changes to a new value but remains constant. This signifies clearly that in spite of varied individual relationship values with process management components, each row of the quality trilogy stages has equal importance to overall quality. The relationship values help process practitioners with a focussed approach in designing value add processes to achieve the best results for the effort.

Considering the relationship scores (columns) for each of the process components, the scores for Process Design and Process Improvement have 13 for each and are highest as compared to others (Process Strategy = 9, Process Implementation=7, Process Control= 9). This indicates that process design and improvements need more focused attention towards quality trilogy elements during process definition and modelling. This analysis clearly reflects how and to what extent the Juran Quality trilogy elements are important in effective formulation of the processes to meet the organizational and business needs. The degree of association between quality and process component pairs is explained in the following section.

5.2 PQR Table

The association of quality and process management evaluated based on their interactions and explained in the below table:

Quality and Process components	Degree of association	Description
Quality Planning and Process Strategy	Moderate	Decisions on process identification and selection are primarily driven by the business needs and high level customer requirements. The requirements are not matured enough for detail quality planning while formulating process strategy.
Quality Planning and Process Design	Strong	Process objectives are formulated and aligned with the quality objectives which are derived from customer and business needs and processes are designed by defining the means to meet those objectives.
Quality Planning and Process implementation	Moderate	The defined processes are implemented and performed to meet the planned process and quality objectives. Process implementation should ensure achievement of these objectives.
Quality Planning and Process Control	Weak	Outcome of process monitoring and control trigger the actions for bringing back the process on track in case of variations but process and quality objectives not change unless there is a change in customer and business needs.
Quality Planning and Process Improvement	Strong	Improvements are initiated to enhance the process maturity level based on performance results and accordingly the process and quality objectives revisited in line with customer and business needs.
Quality Control and Process Strategy	Weak	The feedback on effectiveness of performed processes in achieving the objectives triggers for process improvement initiatives but not revisit of process strategy.
Quality Control and Process Design	Strong	Appropriate process measures for tracking and evaluating the actual process and quality performance are addressed during process design. These measurements trigger the actions against the deficiencies between expected and actual performance.

Quality Control and Process implementation	Moderate	Effectiveness in process implementation is measured in terms of achievement of process and quality objectives. Effective implementation is ensured by taking appropriate corrective and preventive actions.
Quality Control and Process Control	Strong	The identified Process and Quality control measures are complimentary to each other and evaluated against the process and quality objectives during the course of process execution.
Quality Control and Process Improvement	Moderate	Analysis results on outcome of the Quality control activities are the sources for the potential process improvements.
Quality Improvement and Process Strategy	Strong	Identification and initiation of specific improvement projects to enhance the operational efficiency provides inputs for development of new processes and revisit of organization process strategy.
Quality Improvement and Process Design	Moderate	The planned process and quality improvement objectives addressed while process designing along with appropriate performance measures.
Quality Improvement and Process Implementation	Weak	Beneficial change oriented processes implemented through systematic quality improvement plan. A contribution towards quality improvement during process implementation is very low.
Quality Improvement and Process Control	Moderate	The revised process and quality objectives evaluated and controlled through appropriate corrective and preventive actions.
Quality Improvement and Process Improvement	Strong	Both Process and quality improvement are complementary with each other and contribute equally in achieving organization goals. Both shares inputs and outputs with each other.

6 Conclusion

In this paper, the sources and factors responsible for the software process variation were detailed along with their adverse effects on performance of process. The software process dynamics model was discussed extensively and a process dynamics relationship formulated to provide an understanding of the impact of process variation on overall process stability and intermediate/final deliverables.

The relationship between process capability, control and improvement was presented to demonstrate that incremental change in process capability is a time dependent function of process control and process improvement. An approach for characterizing the process variations, defining root causes for variation, establishing process control and improvement actions was described.

Also presented was the SDICC approach for process management. This methodology, based on the Juran quality trilogy model, helps process practitioners design new process and improve existing process performance.

Finally, the relationship between process and quality established through a PQR Matrix demonstrated the importance of Juran quality trilogy components and their association with process management components. The degree of association, along with a description of each association is included in a convenient PQR table.

The author feels that the importance and need for addressing quality factors from process inception to institutionalization, as detailed in this paper, can be useful for a process group involved in process definition and improvement efforts.

References

Raymond J. Madachy, *Software Process Dynamics*, John Wiley and Sons, Inc, November 2007

Qing Wang, Dietmar Pfahl, David Mitchell Raffo, *Software Process Dynamics and Agility, International Conference on Software Process,* ICSP 2007, Minneapolis, MN, USA

Jean Claude Derniame, Badara Ali Kaba, David Wastell, *Software process: principles, methodology, and technology,* Springer-Verlag Berlin Heidelberg 1999

Silvia T. Acuña, Maria Isabel Sánchez-Segura, *Series on Software Engineering and Knowledge Engineering Vol.* 18 *New trends in software process modeling,* World Scientific Publishing Co. Pte. Ltd. 2006

CMMI Product team, *CMMI® for Development, Version 1.3*, Software Engineering Institute, 2010

Thomas Pyzdek, Paul A. Keller, *The Six Sigma Handbook: A Complete Guide for Green Belts, Black Belts, and Managers at All Levels,* McGraw-Hill Professional, 2009

Howard S Gitlow, *Quality Management 3E*, Tata McGraw-Hill Education, 2009

Kai Yang, *Voice of the Customer: Capture and Analysis*, McGraw-Hill Prof Med/Tech, 2007

Basem S. El-Haik, Adnan Shaout, *Software Design for Six Sigma: A Roadmap for Excellence*, John Wiley & Sons Inc, 2010

Cornelius Herzog, *Strategic Tools in Dynamic Environments: A Framework,* Grin Verlag, 2010

Besterfield Dale H, *Total Quality Management,* Pearson, 3rd Edition, 2011

Craig Gygi, Bruce Williams and Terry Gustafson, *Six Sigma Workbook For Dummies,* John Wiley & Sons, 2006

International Organization for Standardization, *ISO 9000:2005 Quality management systems -- Fundamentals and vocabulary,* 2005

Michael George, *Lean Six Sigma: Combining Six Sigma Quality with Lean Production Speed,* McGraw-Hill, First Edition, 2002

P N Mukherjee, *Total Quality Management*, Prentice Hall of India Private Limited, 2006

Risk Measurement for the Real (and Imperfect) World

Darryl Nicholson
ContactDarrylNicholson@gmail.com

Abstract

Walk a mile in my in my shoes. Calibrated risk taking is one of your core skills. Time to market is a measure of success. Your product is not tangible but rather a service you provide. Your skills as a QA engineer are primarily measured by your ability to accurately articulate the risk of a production release in a Software as a Service (SaaS) environment. Your clients are fixated on schedules; yet they demand and expect the delivery of high quality software as a matter of course. You are responsible for risk management and release readiness all the while knowing that your QA team's core competency is judged by how well the team gauges what is "just enough" regression testing. In such an environment, how do you define your methodology so that you can determine acceptable risk?

Do you find my shoes comfortable or are you developing blisters? Well, we routinely deliver software which successfully navigates between the financial dictates of swiftness to market and the professional QA engineer's passion for perfection. This paper will outline how our software development life cycle copes with the new QA reality of SaaS. I will specify how we use software tools and procedures to measure, define and mitigate risk. I will outline how we traverse our path to production and balance speed with quality.

"Just enough" regression testing relies on the idea that there exists a "Minimum Regression Set" (MRS) that is quantifiable and that deterministically exercises the System Under Test (SUT). In my paper, I will explain the whys, the hows and specifics of this methodology. MRS is defined and measured per production release and allows you to evaluate the regression risk of any given software deployment at any point in time. Basically, the risk statement is the percentage of executed test cases derived from calibrated set where one measures API and code coverage. This paper outlines the methodologies used to reduce the MRS to smallest set of test cases needed to provide "just enough" testing so you can deliver high quality software as quickly as possible…ahead of your competitors.

Biography

Darryl Nicholson has been VP of QA & Programs at Vesta Corporation for the past five years where he has morphed the traditional QA department into a modern, risk-tolerant culture focused on service delivery. He has the conflicting responsibilities of both Quality Assurance (where he is measured by the quality of the code release) and Project Management (where he is measured by time to market). Fascinated with the business drivers of software development, he has a unique approach to QA. A veteran of four startups (Preside, Oresis, Q-Optics/Elematics, Polyserve), Darryl is a strong Telecom Engineer with solid software QA and sales engineering expertise.

Darryl has a Computer Engineering degree from the Royal Military College (RMC) of Canada.

Copyright Darryl Nicholson June 20th, 2012

1 Introduction

Risk. A simple word that drives an industry, fosters intense discussions, generates philosophical discussions and creates and destroys entire careers. Risk management defies mathematical precision, challenges perceptions and causes endless debates.

This paper is about regression testing and how careful calibration of your product test plan can drive risk management for any given software release. It's about how to define a quantifiable Minimum Regression Set (MRS) that deterministically exercises the System Under Test (SUT). It's about how to develop and articulate a testing plan to clients in a use case language that all stakeholders understand. I propose to walk through our processes and methodologies to illustrate how we manage risk in an imperfect world where everyone has an opinion, wants to review the results and demands that the schedule be shortened!

A complete test plan for a software release can have multiple phases and numerous components. For example: Regression testing, load and performance testing, automated testing, new feature/functionality testing, API certification, etc. This paper is simply about Regression testing and describes a minimalistic approach that seeks to find balance between software quality and the economic reality of testing where every test case run costs time and money. We seek to define what is "Just enough". In this paper, I will explore how components of our software development life cycle help cope with QA in a "Software as a Service" (SaaS) environment. I will specify how we use software tools and procedures to measure, define and comprehend risk as it applies to test case execution in our imperfect world.

2 Background

In our SaaS environment, we sell services that rely on our software. We are not directly compensated for lines of code we deliver but for speed to market of new services and stability of our current production features. Our clients derive a significant portion of their revenue from these services so stability and high quality software are expected. As our revenue model is tied to production services many of the traditional software development models make no sense.

In this environment, one needs to manage risk and deliver quickly to drive revenue. Caution simply doesn't pay. As you cut corners and get lucky time and time again, cultural tolerance for risk grows. One needs a balanced process to keep all stakeholders and key business owners apprised of the decisions being made and the inherent risks being taken. It is easy to become blasé when negative consequences are avoided time and time again with fire drills and luck. However, Mr. Murphy can't be bribed forever. A testing methodology based on MRS quantifies the risks taken for each release and is easy for non-technical decision makers to understand.

Our Production environment is a mix of isolated stacks of dedicated customer software based on a common framework that rely on a Service Oriented Architecture (SOA) to deliver services. This hybrid model was built for speed to market, innovation and meant to handle lots of changes. Unfortunately, complexity increases with every release and the testing time to execute new feature & regression testing continually grows. Testing time is constantly under pressure to reduce and demonstrate maximum test execution in the shortest possible time frame. Since our test case repository was growing exponentially with rapid feature addition, we had a problem that had to be solved.

3 The Problem

Over the past couple of years, we have engaged customers and stakeholders in a formalized test case review process. While client engagement at this level is an overall positive experience, the side effect is continuous test case growth in an unstructured quasi-subjective manner. Additionally, the problem is compounded with additional growth in test cases by ongoing product evolution, new clients, solving for production test escapes and dealing with customer feedback. With each new release, the regression

cycle needs additional time and/or additional resources to complete. Project managers, business executives, marketing and customers never like this answer. The problem continually grows harder as we add new feature functionality across our customer base. Constant innovation creates new test cases that add to the regression base and that demand longer test cycles to release.

Like many software companies, we incur technical debt, make mistakes and consequently have test escapes (aka defects) that require immediate production fixes. Analysis identifies where the test escape occurred and best practice test refactoring results in new test cases to avoid repetition of the problem.

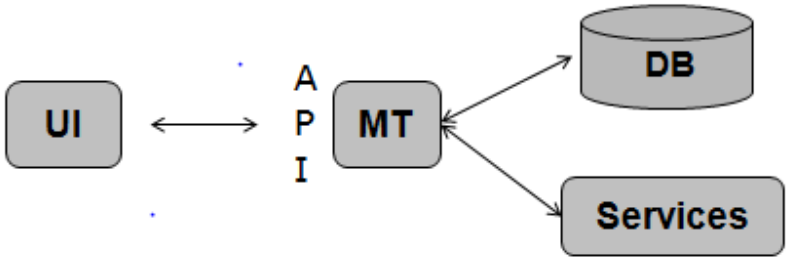
In short, our regression test set grew substantially over time and we found ourselves in a situation where our Quality Assurance Test Management process was neither sustainable nor scalable. Our QA department simply did not have the resources to keep up with the current pace of innovation and resulting test case growth. The increasing maintenance burden represented an ever growing set of test cases requiring ever more time and resources to complete; our problem was becoming unmanageable. After much deliberation, we decided to attack the regression testing problem and instrument our test cases using code coverage techniques. We define the resulting test case set derived from this analysis as the "Minimum Regression Set" (MRS) which is a work product that easily maps to use cases and requirements in a way that all stakeholders can understand.

The MRS philosophy is minimalistic in scope and is very much in sync with the current "Lean Startup" thinking. Reis argues in his book "*The Lean Startup*" (Reis, 2011) that all startups need to get a Minimum Viable Product (MVP) to market as fast as possible to iterate and learn from the market place rather than strive for perfection. The MRS is our QA product equivalent and, like Reis's philosophy, we want to get our MVP into use as fast as possible since time is money.

4 The Environment

Our client base primarily exists of large Telco carriers who, for the most part, choose to operate a fairly normal waterfall methodology. We operate internally in a very iterative mode with a core focus on automated unit tests that drive API & code coverage in order to meet our clients' needs and aggressive timelines. Our development team embraced the Test Driven Development (TDD) methodology for Java (Koskela, 2008) and has an extensive set of Unit tests that drive our Middle Tier (MT) API's:

Figure 1: Three Tiered Architecture



UI = User Interface Layer [Many different applications; mainly Java based]

MT = Middle Tier Layer [all Java]

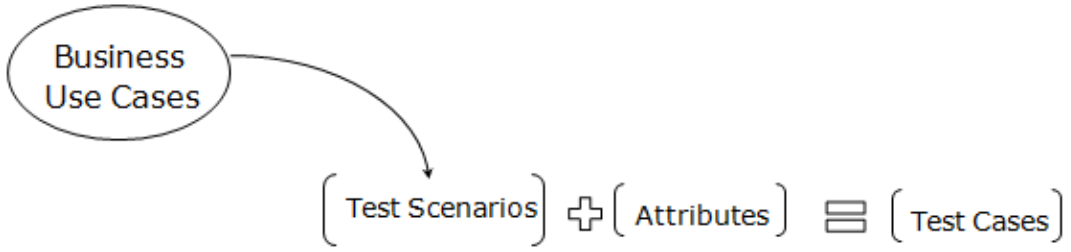
DB = DataBase

As a direct result of this methodology and running unit tests, an instrumented Middle Tier is available for all feature complete QA Releases. The code coverage tool chosen was Cobertura which provides a rich data set. Cobertura is an open source application that instruments the Java bytecode after compilation and works well with our standard eclipse IDE.

5 Test Scenarios

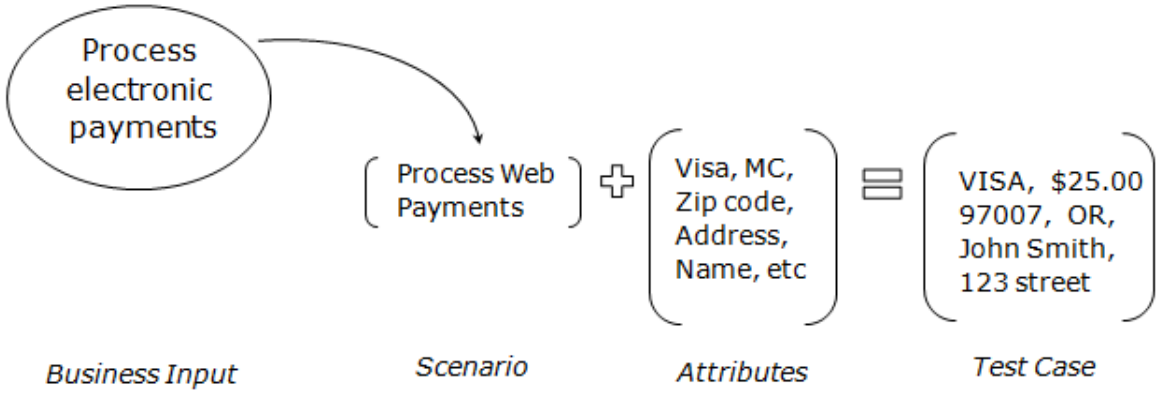
Given our clients have a tendency to describe needed changes more in terms of business use cases, marketing ideas or product delivery strategies rather than traditional software requirements, we need to use these inputs as our starting point. We take Client definition, in whatever form it arrives, and use this to describe "Test Scenarios" that are focused and mapped to client inputs. We purposely segregate out the test case data and refer to these elements as "Attributes". This methodology allows us to iterate on the Test Scenarios at a high level while maintaining their causal relationship to the client input and keep the number of test artifacts we need to manage and review to a reasonable set. Our process works like this:

Figure 2: Scenario Definition Model



For example, as an electronic payments company we process credit card transactions on many different UI channels from every state for multiple different payment devices and amounts. Using our analysis methodology above an example would be:

Figure 3: Example



Once we have a proposed set of Test Scenarios and Attributes, we can review these with all interested parties and stakeholders (both internal & external to our company). Reviewing and generating Scenarios & Attributes in this subjective manner with clients and key stakeholders is somewhat inefficient and certainly creates lots of redundancy due the non-technical nature of the analysis. However, collaborative review up front when the project request is still fresh in everyone's mind creates joint ownership of all the needed Scenarios from a subjective point of view.

A typical review process for one of our web products will create 700-900 unique Scenarios. Now the question remains are all defined Scenarios truly needed?

6 Test Calibration

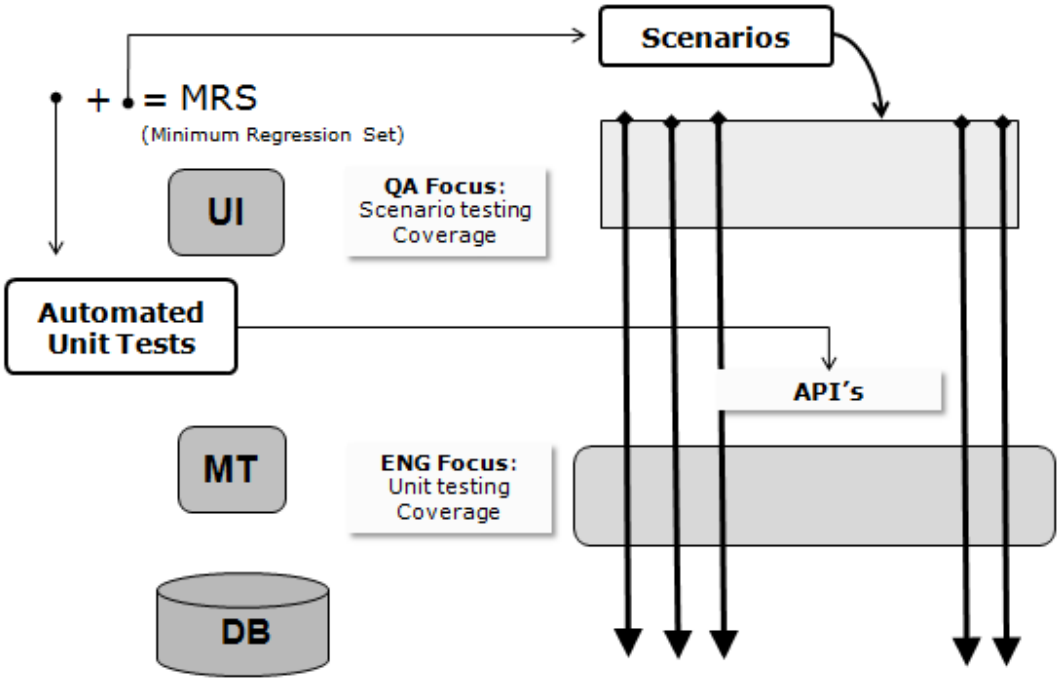
Now that we have collected a large set of Scenarios that we intellectually believe represent the needed set of regression test cases to cover the feature set we need to calibrate the list. "Test Calibration" is simply the term we use to create the MRS using code coverage analysis. Our goal in this effort is to classify all the defined Scenarios into three categories:

- Category 1: The MRS. A single Scenario that exercises a unique path of code though the system, is repeatable and can be measured as unique with Cobertura.
- Category 2: A Scenario that does not add code path uniqueness, therefore is not part of the MRS, but adds a significant & unique data set based on one of the defined attributes.

- Category 3: A Scenario that has neither code path uniqueness nor adds a unique Attribute to the data set. For most part this category represents the emotional category of Scenarios and can usually be traced to previous production test escapes, emotional feature sets and the illusion of importance.

Defining the MRS is actually a simple process where the QA Engineer takes the instrumented Cobertura build which the engineering team built to run their unit tests and replaces the deployed MT-JAR file in the system under test. Then it is a simple manner of individually running each Scenario and validating that code coverage increases with the execution of the scenario. If code coverage numbers do not change, then the Scenario is a candidate for Category 2 or 3. The methodology we use to determine the difference between these two categories is outside the scope of this paper. However, it involves a very similar process where we graph unique attributes and their data sets by Scenario to illustrate uniqueness and thusly define Categories 2 and 3. Extending the 3-Tiered Architecture diagram from figure 1 to illustrate this process looks like this:

Figure 4: Test Calibration



Cobertura provides detailed and simple to understand code coverage data. As you can see from this example (report example from the cobertura home page on sourceforge) the tool dynamically tracks coverage and it is a simple matter to see if your coverage numbers increase as you complete a Scenario.

Figure 5: Cobertura Code Coverage Example

Coverage Report - All Packages

Package	# Classes	Line Coverage		Branch Coverage		Complexity
All Packages	55	75%	1625/2179	64%	472/738	2.319
net.sourceforge.cobertura.ant	11	52%	170/330	43%	40/94	1.848
net.sourceforge.cobertura.check	3	0%	0/150	0%	0/76	2.429
net.sourceforge.cobertura.coveragedata	13	N/A	N/A	N/A	N/A	2.277
net.sourceforge.cobertura.instrument	10	90%	460/510	75%	123/164	1.854
net.sourceforge.cobertura.merge	1	86%	30/35	88%	14/16	5.5
net.sourceforge.cobertura.reporting	3	87%	116/134	80%	43/54	2.882
net.sourceforge.cobertura.reporting.html	4	91%	475/523	77%	156/202	4.444
net.sourceforge.cobertura.reporting.html.files	1	87%	39/45	62%	5/8	4.5
net.sourceforge.cobertura.reporting.xml	1	100%	155/155	95%	21/22	1.524
net.sourceforge.cobertura.util	9	60%	175/291	69%	70/102	2.892
someotherpackage	1	83%	5/6	N/A	N/A	1.2

Report generated by Cobertura 1.9 on 6/9/07 12:37 AM.

While our stated goals for the MRS are 100% API and 100% code coverage, the reality is that 100% code coverage is not realistic. There are numerous reasons why a particular Class / Line or Branch can be

ignored safely and as long as the Dev Lead and QA Lead agree we mark these as exceptions on our internal Wiki where coverage and MRS data are kept.

Cobertura provides lots of coverage detail. The ability to drive down from a package level summary to the Java Class and then finally the actual source allows for complete inspection of behavior of a Scenario in the MT layer. Furthermore, the Cyclomatic complexity gives an easily understood scale (bigger the number – the more complex the code) and identifies areas where code reviews should start.

The MRS creation activity yields several interesting results. Generally after execution of approximately a third of the defined Scenarios, the code coverage needle will stop incrementing but we will have nowhere near 100% coverage. This is the moment where we realize that the Scenarios analysis done as an intellectual exercise has missed a number of valid cases. Typically what is missed and overlooked are the error handling routines, obscure use cases and available functionality that was not obvious at review. When running with code coverage enabled, these potential test escapes are very obvious.

7 User Interface White Space

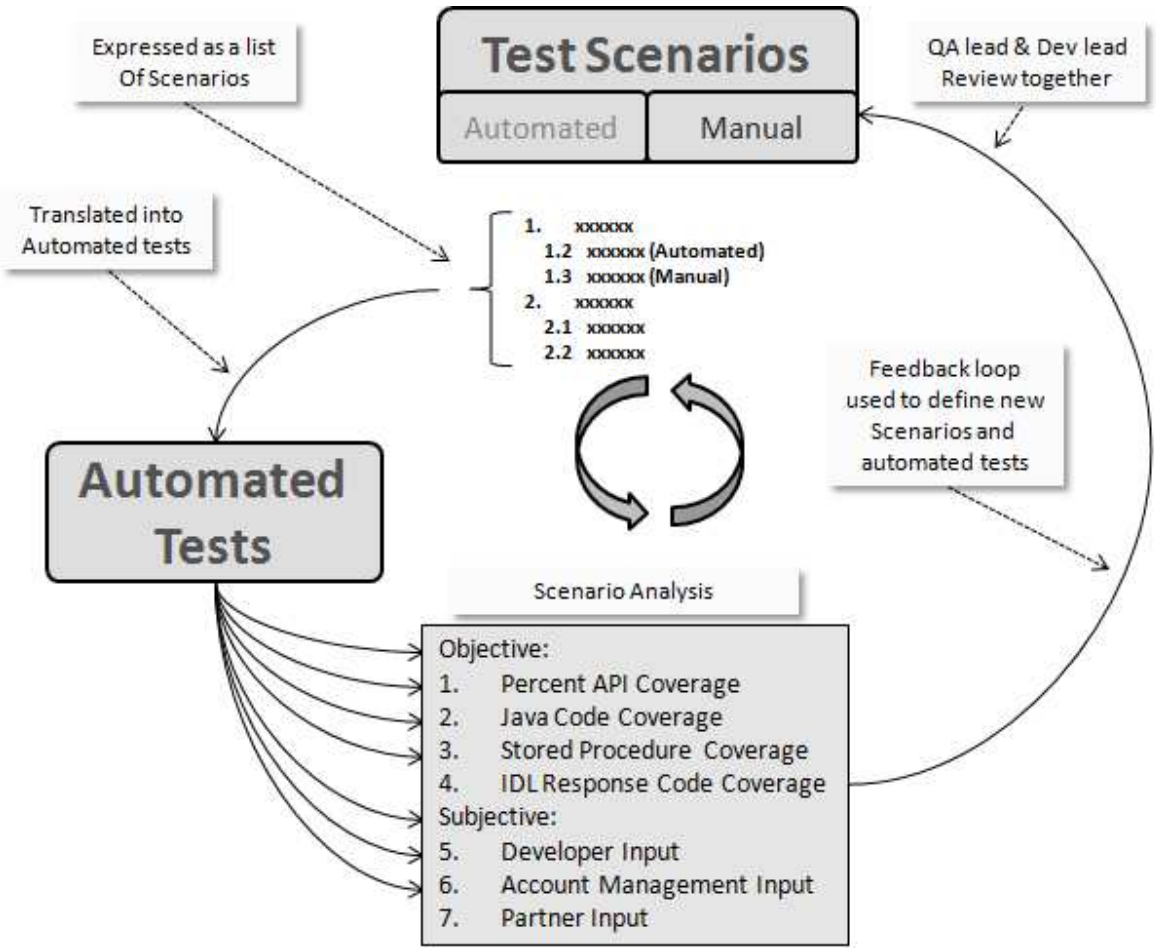
The methodology outlined so far where we define Scenarios and measure code coverage in the middle tier simply implies code coverage in the UI layer. We have made a conscious decision not to directly measure and instrument the UI layers in our system because of the rapid rate of change these interfaces experience. The economic reality is the return on investment of this activity is not warranted due to the time and effort required to create an MRS and frequency of changes.

Once the MRS is defined for an application, one final code review is needed of the UI layer. It is the intention of this review exercise to define what we refer to as the “White Space”. The White Space is the UI code structures not exercised by any of the Scenarios since their scope is entirely contained inside the UI framework. While this activity is code review based, we add these items to the MRS anyway. We label these Scenarios this way since they live in the White Space of Figure 4. Some examples of these items that generate additional White Space Scenarios: JQuery elements, Analytic web tags and form based validation logic.

8 Scenario Lifecycle

Now that we have defined the MRS, let’s examine the overall lifecycle. The diagram below shows the highly iterative nature of our definition process. For each major release where we add new feature / functionality or where we undertake a significant re-factor to our code base, we have a pragmatic process to define or update the MRS for any particular application in our system.

Figure 6: Scenario Life Cycle



A feedback loop ensures we capture needed automated tests and that new Scenarios are captured in the "Test Scenarios".

9 Test Escapes

They happen. Despite best efforts and rigor in the analysis phase to determine what is needed for regression as the MRS, mistakes happen and defects make their way to production. When these events occur, the root cause analysis of the test escape can now be expressed as a part of our methodology and in terms of MRS. In our system, test escapes tend to occur from one of the following cases:

- Automated test failure where we have misinterpreted an expected exit condition; a coding failure in the test case that gives a false positive or fails to run properly.
- MRS definition inaccuracy where an important Scenario is missed in the "White Space".
- The review process overlooked the importance of a particular set of code and a mistake was made identifying it as an acceptable non executed code path.
- Scenario was not executed because of time & resource constraints.

In the first 3 cases, test escapes are easy to resolve by defining new Scenarios & automated tests to ensure the issue doesn't happen again. In the fourth case, this is simply the price we pay for a too aggressive schedule.

10 The Deliverable

At the end of the day, Test Calibration and the resulting definition of the MRS is about risk management, speed to market and delivering high quality software in the context of an economic framework. We want to define the smallest set of regression tests that need to be run in the shortest amount of time because running test cases costs money and delays service delivery.

This structured approach works and allows for precise discussion around the fit for release of a particular Release Candidate build. This is invaluable data for the Project Manager and the "Go/No-Go" meeting to describe precisely in business terms the readiness of a release. Figure 3 is an illustration of how each Scenario can be described in a simple line format, Cobertura code coverage details easily fit on a slide deck and the net result is a simple presentation that clearly outlines the release status. Even more powerful to this discussion is the fact key stakeholders were engaged in creating the initial draft of the Scenario list. The systematic reduction to the MRS is a powerful definition of "Just Enough" testing and facilitates a conversation about test completeness. Clearly, the regression risk of a production release at any point in the project plan is simply the percent complete of testing the MRS; a simple one line sentence that conveys product readiness.

We live in an imperfect world. Should a date occur and business drivers dictate that a production release needs to go live when QA has not finished testing then QA has a simple message for the team → MRS = 45%.

References

Koskela, L 2008 *Test Driven, Practical TDD and Acceptance TDD for Java Developers.* Manning

Reis, E 2011 *The Lean Startup.* Crown Business

Thomas, Young, Brown, Glover 2004 *Java Test Patterns*. Willey

http://cobertura.sourceforge.net/index.html (last accessed June 2012)

www.eclipse.org/ (last accessed June 2012)

http://en.wikipedia.org/wiki/Cyclomatic_complexity (last accessed June 2012)

Testing Site Personalization, Unpacking Personalized Sites Through Emulation

Amith Pulla
Intel Corp.
amith.pulla@intel.com

Abstract

Site personalization is becoming one of the most effective characteristics in online marketing websites. In a competitive environment, a few seconds matter if a site visitor is finding what they are looking for. Site personalization helps deliver personalized content, promotions and experiences. Though a very powerful solution that helps both the users and business, it brings new challenges to people who are testing this capability and to content authors trying to validate their content before publishing.

One of the widely used techniques for personalized content experience for different user groups is to install a client side cookie to track user location, origin, interests and navigation to the target site. The client side cookie will determine the content or pages that will be rendered to the user. Multiple versions of the same page are developed and tested individually by content authors and designers, but there is no easy way to test the navigation, page rendering and user experience in an efficient fashion.

This paper tries to address this problem with an efficient solution by developing a small cookie manipulation tool that the tester can use to change the cookie settings. With this approach, the tester can easily use the web tool to set the cookie based on the experience being tested and test the user experience and content for target audience.

Biography

Amith Pulla is a senior software quality analyst at Intel Corp, currently working at the Intel site in Hillsboro, Oregon. Over the last decade, he has been involved in software testing strategies and processes for applications mainly focused on sales and marketing. Amith has worked extensively on projects involving multiple platforms and complex architectures. Amith also worked on developing test methodologies and techniques to meet the business needs and timelines. As part of his QA lead role, Amith focused on improving and refining QA processes and standards for efficient software testing.

Amith has an M.S. in CIS from the New Jersey Institute of Technology; Amith got his CSTE and PMP certifications in 2006.

1. Introduction

Site personalization technology can dynamically insert and customize content based on individual user profiles. The user profile can be determined by user's implicit behavior, browsing patterns and preferences, and sometimes by explicitly captured details provided by the user.

Both implicit and explicit information for a specific user is used to personalize the content on the site and pages. Implicit information can be derived from user's location, navigation patterns, search terms used or links clicked. Explicit information is something user declares as part of the navigation, this could be user providing information regarding age and gender, or in corporate marketing context the user can provide information related to company background, size, role and product development interests.

Personalization is becoming very popular with e-business and marketing sites by providing special treatment to the site visitors in the form of relevant content and information. One of the key goals of personalization is to attract and retain visitors that eventually lead to better user experience, satisfaction and sales. In big corporations, personalized sites for employees, vendors and contractors can improve productivity and customer satisfaction by simplified and faster access to data, information and applications based on role and profile.

Site personalization gives an edge to companies by delivering sites and applications that are easier to use as they are tailored to meet individual needs. Several studies proved that personalization increases conversion rate and revenue metrics. Personalization includes collecting and storing information concerning site visitors and analyzing the information, (Willy Chiu 2001). Based on the processed information, the site delivers the right content, experience and features to each visitor. Personalization techniques can enable the site for target advertising, promote products, create relevant and meaningful interactions across multiple channels, recommend articles and provide suitable guidance.

Designing personalized sites can be complicated; it involves a thorough process of gathering and storing information about site visitors, both on the client and server side, then based on the information the server needs to deliver the right content to each visitor. In marketing sites, personalization can enable companies to aim advertising, promote products, and recommend articles and reviews.

2. Unpacking Personalized Sites

To better test the personalized sites in a test environment, we need to understand how personalization variables drive the content on the page; later we will learn how to unpack personalized sites using the emulation tool as a testing technique. Unpacking a personalized site means opening up the site to view all versions of content for each module and page based on each combination of personalization variable values. Personalized sites cannot be fully tested unless they are unpacked to view all possible behaviors and facets of the site.

Personalized sites are often driven by a personalization engine in the application server layer (middleware) of a multi-tier architecture design, rendering the personalized content based on available data for preconfigured variables. The personalization engine works behind the scenes to render relevant or targeted content to the end user with the personalization logic and rules that are invisible to the users. Personalization variables play a key role in content rendering for a specific user; these variables are set in the personalization engine based on user interaction with the application or website. Often separate complex algorithms are used for each product line to support different personalization techniques. As Aggarwal, Wolf, Wu, Yu said (C. Aggarwal, J.L. Wolf, K-L. Wu and P.S. Yu 2000), "Web personalization algorithms are fast becoming an essential component of e-commerce". These algorithms use personalization variables as inputs and process them to customize the content. There are many different types and categories of personalization; we will focus on four here that are popular in the industry for sales and marketing sites.

2.1 Implicit or Contextual Personalization

Contextual personalization uses implicit profiling techniques. This personalization is mostly capturing information on where the user is coming from and what the user is looking for by using search or clicking specific links. If the user is searching using specific keywords on the popular search engine sites, the site personalization engine can capture the keywords used by the user and serve up relevant content and products once the user lands on the target site.

2.1.1 In an automobile industry example, if the user searches for most fuel efficient car on the web using popular search engine sites, the specific car manufacturer site will be shown in the search results. Once the user or the potential buyer clicks on car manufacturer site link, the user enters the site and the site can highlight or recommend their most fuel-efficient cars from their entire line up of cars. This ensures the potential buyer is looking at the product offerings that he or she is most interested in first.

2.2 Explicit or Profile-Driven Personalization

Profile-driven personalization uses explicit profiling where the user provides some identifying information that can be used to personalize content based on user needs. Profile-driven personalization is all about the information that a user provides to the website during the current or a previous visit, either through profile information or through online questionnaire. The profile information can include user's job function, interests, age, gender etc. Once the profile is created online, for each subsequent visit by the user, the saved information is used to render personalized content.

2.2.1 In a sporting goods online store example, a user creates an account in the initial visit and provides some basic information about them, like user's age, location, favorite sports, gender. In the following visits to the site, the personalization engine can provide content and recommend products and offers based on users' interests, preferences and previous product views or purchases.

2.3 Personalization by eNurturing and Cultivation

eNurturing refers to the specific sales strategy developed by a company based on customer's position within the sales cycle. Depending on the customer's position and purchase history and trends, specific online activities, promotions or products are pushed to a user or visitor. The website content is changed to align with sales cycles to encourage favorable decisions. eNurturing strategy is widely used in business-to-business (B2B) marketing. eNurturing creates several touch points with the potential customers and at each touch point a different content is pushed to the potential customers either in emails or online content. Cultivation is similar to eNurturing; it's a sales strategy to develop relationship with prospects over a period of time by pushing content at different communication points in the sales cycle prior to the purchasing decision.

2.3.1 A good example of eNurturing is IBM's marketing of their WebSphere software solutions, (Karen Gedney 2003). IBM's Worldwide Direct Marketing of WebSphere group adopted an eNurturing strategy that took a high-value approach to intelligently move potential customers through the sales cycle. There are several touch points created, starting with online registration, invitation to webcasts and interactive ROI assessment. The registration process provides a value exchange for prospect's time investment. Instead of providing same standard technical content to each prospect, based on the profile information provided by the prospect during the registration, the personalization engine creates a complete business information kit. The kit includes a product spec sheet, a customer case study, a demo, and a research paper from a well-regarded IT advisory firm. On subsequent visits to their site, several touch points are used in a sales cycle to deliver relevant content to encourage favorable purchasing decision.

2.4 Score-Driven Personalization

Score-driven personalization rates the customer's inclination to make a decision on a purchase. Depending on the user's behavior, the personalization engine can determine when the customer is likely to make a decision to buy a product or service, and the site can provide higher level of service to the

customer, through specific promotions and discounts. The application scores all users or potential customers and renders content based on score.

2.4.1 Score driven personalization is used commonly in direct marketing. A good example is how the Dutch bank ING's marketing campaigns used this approach to increase average campaign response rates and at the same time reduced direct marketing costs, (Alexander Hesse 2009). ING implemented a centralized campaign management program that creates personalized offers and content in real time. The centralized campaign management system uses historical and recent customer interaction data to score different customers and based on the score generated, the application creates real-time next best actions (NBAs) for each customer that include individualized product offers and promotions.

3. Cookies in Personalization

Browser cookies on the client side play an important role in site personalization; the personalization engine sends the browser cookie(s) to the client machine based on the user preferences and personalization variables. The cookie on the client side drives the user experience.

In implicit profiling, as part of the initializing process, the user context is set and a cookie is sent from the server to the client. The cookie contains all the user preferences, mostly in encrypted form and drives the content on the pages.

In explicit profiling, a user selects their preferences by entering information in a web form or logging in the application with an existing profile on the server. The personalization engine on the server encodes the preferences in a cookie and sends the cookie back to the client's browser.

After the initialization process, where the cookie stored in the client has all the information about user preferences and personalization variables in an encoded format, relevant content is rendered on the page. In each subsequent visit by the user, the cookie on the client is read by the server and the information is used to deliver relevant personalized content to the user or potential customer.

4. User Context and Initialization

Setting the user context is an important aspect of both the implicit and explicit profiling; user context is typically set by initialization process. As soon as the user lands on the personalized site, the personalization engine tries to capture information to set the variables. Once the variables are set, a cookie is sent to the client and this cookie has all the information in an encoded format on user preferences and interests in the form of variables.

4.1 Initialization in Implicit Profiling

With implicit profiling, the initialization process starts when the user first comes to the personalized site, either from a search engine or by clicking on link in a marketing email or webpage. This process initializes the user context variables such as product or services interests, preferences, location based on available information from user actions. For example, if the user came from a search engine site, the personalization engine sets the variables based on the search keywords used. If the user came to the site by clicking on direct marketing email, relevant email campaign information is used to set variables.

4.2 Initialization in Explicit Profiling

With explicit profiling, the initialization process is straight forward as the user explicitly states his or her preferences by selecting options on the site, by completing an online questionnaire or by creating a profile. Based on the selections made by the user, the personalization variables are set.

4.3 Triggers

Based on the user's behavior and navigation on the website, specific triggers are used to re-set some of the personalization variables. User context is set initially, but if user's preferences change during the session, specific triggers are used to track and update variables. The specific user actions include, profile updates, browse to different product lines, and searches. Once the variables are updated or reset, the information in the cookie changes and the new information is used by the server to drive user experience and content.

For instance, in the online sporting goods store example mentioned earlier, if the user buys products related to tennis, the product interest variable is set to tennis and more tennis related offers and promotions are shown and recommended to the user. Later on, if the same user searches for soccer products and browses soccer product pages, the search or browse actions are used as triggers to reset the product interest variable to soccer and soccer related products are highlighted or recommended to the user.

On Intel's embedded design site, the site recommends targeted technical papers and articles to visitors based on user preferences in the "Recommended for You" module on the home page. Examples of a personalized module are shown below in Figure 1 and Figure 2. Figure 1 shows the "Recommended for You" module with content targeted for a technical audience with technical specifications and white papers. In contrast, Figure 2 shows the "Recommended for You" module targeted towards a general audience with no product preference.

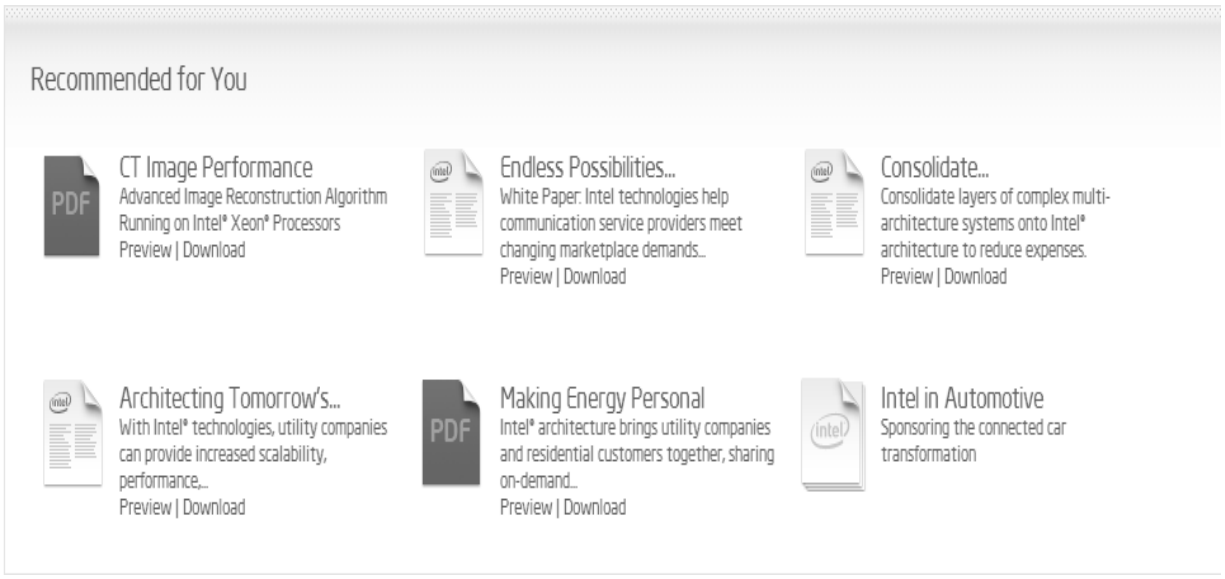


Figure 1

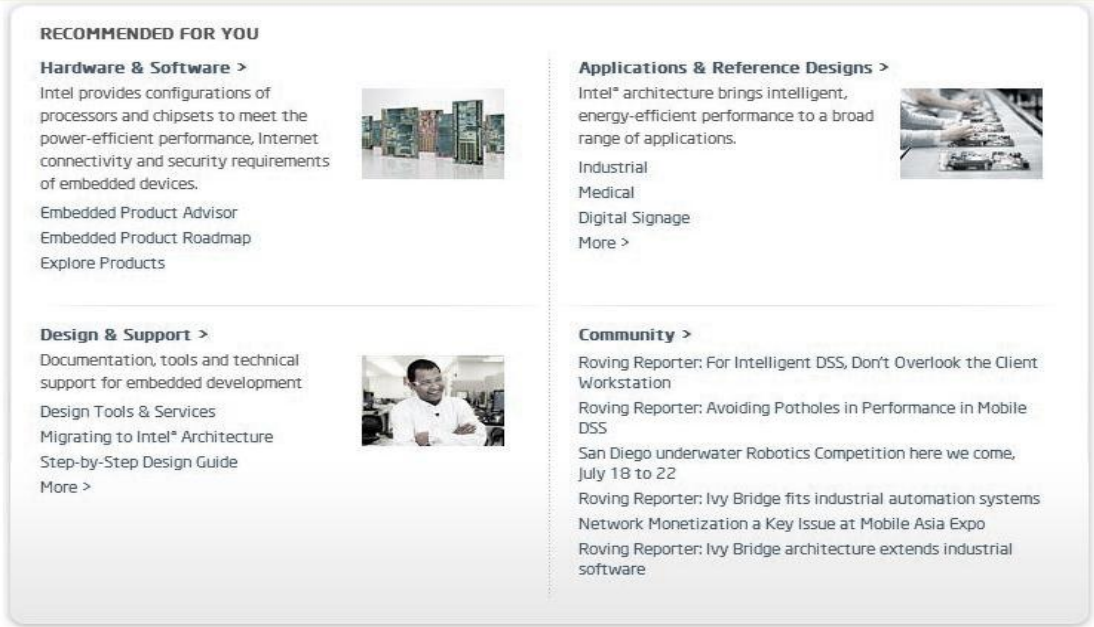


Figure 2

5. Testing Challenges

5.1 Site personalization is a powerful feature, highly regarded by both businesses and visitors. For the people testing the site in a test environment, it can be a nightmare. Site personalization introduces new challenges on how to test the site and cover all content scenarios. Both testers and content authors have challenges in testing content and user experience for each profile and personalization variable. There is no fast and effective method to test websites/web pages without fully going through initialization process, setting the user context and simulating the real life user actions and behaviors. Simulating user behavior and actions is a time consuming and labor intensive process, costing too much testing effort and time. The challenges slightly vary for sites using implicit and explicit profiling, but each requires a number of user actions and behaviors to fully test the content personalized for each variable.

5.1.1 With implicit profiling, depending on the personalization variables that the application or website is tracking, it may take many days of testing to simulate all possible scenarios. There can be multiple user actions and behaviors that the tester needs to simulate for the site to determine and set personalization variable, including how the user got onto the target site, what search keywords were used, which products the user is viewing, specific browsing patterns and the location.

5.1.2 In the context of automobile industry example we talked about in section 2.1.1, for the tester and content author testing in a test environment, they need to search using multiple keywords like fuel-efficient cars, high mpg, hybrid, low maintenance etc. or for a different variable the search terms can be AWD, 4x4, 4WD, cross country etc. to get into the site with specific user context. This searching using hundreds of keywords can be time intensive and makes the testing effort inefficient.

5.2 With explicit profiling, the application or website already knows some information about the user's preferences; this information is provided by user either by creating an online profile/account or data provided by user as part of self-identification and preference settings. As with implicit profiling, for testers and content authors, it may take many days of testing as the testers need to either keep updating the user settings and login/logout of the application or create multiple test accounts with different profiles for each user preference or personalization variable. Both of these methods are time consuming and not efficient.

5.2.1 In the sporting goods online store example we discussed in section 2.2.1, testing the site requires constant updating of the user setting for each age group, gender, location and favorite sport to test the site for all possible versions, again not an efficient effort.

5.3 Because testing personalized sites with implicit or explicit profiling can be challenging and time consuming, we need to explore other tools and techniques to test content and user experience efficiently, shortening the testing window without compromising quality.

6. Emulation as a Testing Technique

Emulation is a widely used concept in software industry for software quality testing and production support to imitate users. The concept behind emulation is navigating a site or application as a user with specific roles or access, mainly to test application behavior, troubleshooting issues and to reproduce defects. While emulation is not a new technique to mimic a logged in experience where a user already has a profile setup, the use of emulation for testing personalized sites with or without login is fairly new.

For testing a personalized website successfully, first we need to unpack the website based on each personalization variable. The testers should be able to validate the user experience and content for each variable or each combination of variables, this will ensure comprehensive content coverage. The emulation approach enables the testers to unpack the personalized site in a structured way to effectively and efficiently test the website.

6.1 Manipulation of Cookies on the Client

The role of cookies in site personalization was discussed previously. In this section we will focus on creating, modifying and manipulating cookies on the client side without user context initialization process. The goal is to trick the server into thinking that the initialization process already occurred and the cookie has been sent by the server. Bypassing the initialization process enables the testers of the personalized sites to navigate the site and test content and user experience for each preconfigured user profile with combination of personalization variables without having to do the actions needed for initialization process. A web tool was developed to create and modify client side cookies based on tester's inputs and selections.

6.2 Setting Personalization Variables Using the Tool

To efficiently test site personalization, the approach we used is to design and build an emulation tool. The goal of the emulation tool is to imitate the user behavior and actions by simply creating and modifying cookies with the personalization variables set, on the client side as if the cookies were created by site's application server. The tool captures the values for the personalization variables or user preferences in a simple UI with a dropdown for each personalization variable. The tool provides a dropdown list for each personalization variable, in our case the variables are the user tier, industry, preferred brand and preferred platform. The testers and content authors validating the page content select the personalization variable values from the dropdowns and clicks on submit. Based on the selected values, the tool creates the cookies on the client side that drives the content on the website. If the cookies already exist, they are overwritten by the tool. The project development team designed and developed the tool to help save the QA team weeks of testing time.

6.3 Tool Overview and Features

The tool has a single screen interface as shown in Figure 3, where the testers and content authors can select the combination of personalized variables and click Submit. Based on the selected personalization variables, the sites content is changed, giving access to all versions of the same page.

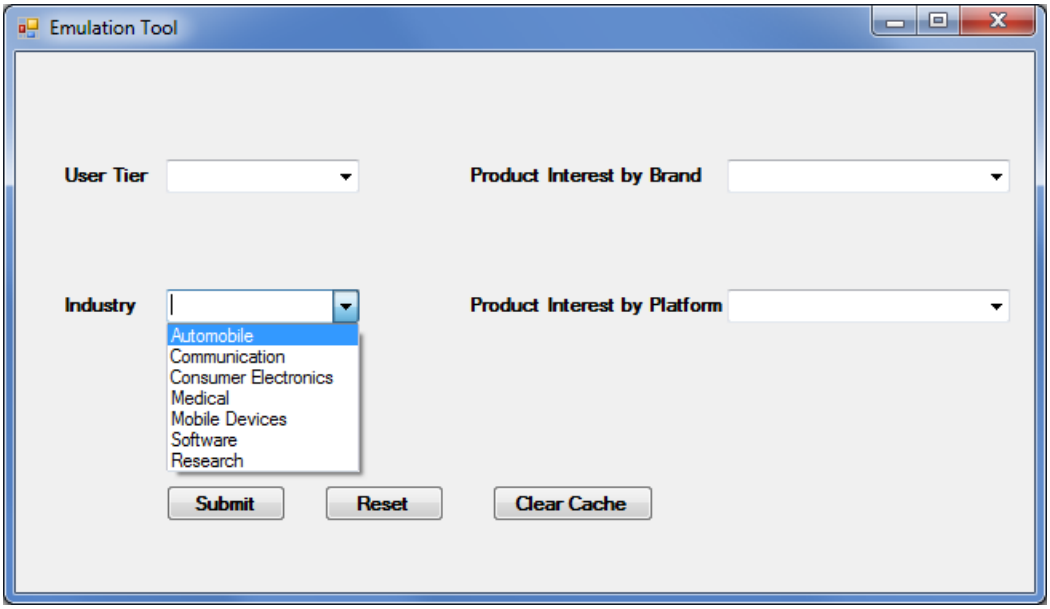


Figure 3

The tool forces the website or the application to render relevant content and pages based on the variables set by the testers. Testers or content authors can navigate the site as if the real user is navigating, which helps the testers identify any issues and defects with content or user experience.

6.4 Testing the Emulation Tool

The emulation tool is a small application, independent of the personalized website. The emulation tool needs separate testing to ensure correct functionality, that the data in UI is setup correctly and the right cookies are generated by the tool based on variable values shown in the UI. The testing effort is relatively small, 2-3 days of manual testing. The testing mainly involves testers manually verifying the information in the cookie that is generated by the tool based on the selected values.

6.5 Limitations of Emulation Approach

Though the emulation approach unpacks the site quickly for easy content validation by circumventing the whole initialization process, there are some drawbacks to this approach. This method or approach doesn't test the functionality related to initialization process and user actions/behaviors resulting in creating a cookie by the server on the client side. The initialization process and cookie creation process by server should be tested separately. The emulation approach is really effective when the site has multiple personalization variables and where each variable can have multiple possible values. Though the emulation approach provides the ability for testers to quickly change the combination of personalization variable values and validate content to each combination, it's not a comprehensive test strategy and additional testing is necessary to validate initialization and cookie creation process.

7. Conclusion

The adoption of this emulation approach helped the QA team and content authors quickly access different personalized version of the same page/URL, without actually recreating the user behavior and actions. The emulation approach bypasses the initialization process and gives testers quick access to personalized content for each profile, saving time and enabling the QA team to complete testing in one third of time compared to the conventional methods of testing, which required simulating user behavior and setting user context. With this emulation approach, several versions of same page or module can be accessed quickly, resulting in faster validation of the entire site and content as well as site's

personalization code. The testing team was able to access and test all personalized pages and capabilities easily, ensuring higher quality of the application and content at the same time shrinking the testing schedule.

In summary, with this emulation approach/solution, the testing team could complete site personalization testing and content validation faster, ensuring higher quality and accuracy of personalized sites with minimal testing cost and effort, and delivering customer satisfaction and quality.

References

Web site personalization by Willy Chiu IBM Technical Library, April 2001

The Intelligent Recommendation Analyzer," by C. Aggarwal, J.L. Wolf, K-L. Wu, and P.S. Yu, to appear in proceedings of ICDCS Workshop on Knowledge Discovery and Data Mining, April 2000.

IBM High-Volume Web Sites, Design for Scalability, December 1999

E-Nurturing, the IBM Way by Karen Gedney, April 2003

Case Study: ING Delivers Personalized Product Offers across Channels in Real Time by Alexander Hesse, October 2009

Testing in Production: It's not just for services anymore!

Adil A. Gheewala – adilg@microsoft.com
Sam Bedekar – sbedekar@microsoft.com
Weifeng Yao – weiyao@microsoft.com

Abstract

The test team in an organization often has a large set of new and regression tests to execute to ensure the quality of the features. The ever growing test matrix is another strain on the test organization as it needs to support multiple legacy clients, release after release.

Testing in production where real users test real scenarios is a great way to harness the power of the crowd, get a more end user centric test coverage, and reduce some of the tests the actual tester needs to perform. End users would install the application and use it in their normal mode. The TiP system collects the relevant user interactions and allows the owner to make decisions based on the data – feature test coverage and/or feedback on the feature.

The Lync Telemetry v-team has developed a tool called Scenario Compass which listens for events from Lync and any other Office 2013 applications to distinctively identify the set of actions/features a user exercises while using them. This allows the feature area owner to have data behind his crowd sourced tests. Scenario Compass also collects machine and application specific data which can be used to check off configurations in the test matrix. This allows the test team to focus their efforts on configurations with low coverages. Scenario Compass also flags failing cases. If a certain anticipated event does not occur in a given amount of time, that scenario is flagged as a failure. Using the failure analysis component of Scenario Compass, one could also do performance tests in the real world.

Scenario Compass has been used in the Lync client team to optimize the test passes by identifying areas covered by internal Microsoft users and reducing effort in them and identifying holes in coverage/configurations and focusing more in them. We also have built games into the tool to increase adoption and more importantly influence users to execute a certain set of features.

This paper, describes some of the challenges we faced in our testing, how we overcame them, the solution we built and how others can leverage those in their environment.

Biography

Adil Gheewala is a Senior Test Lead at Microsoft Corporation, who joined Microsoft in 2002 and worked in Windows for 4 years before joining the Lync client team. Adil is passionate about optimizing test methodologies & processes, and driving complex features to high quality.

Sam Bedekar is a Senior Test Manager at Microsoft Corporation in the Lync team. He has worked in UC space for ten years. Sam has a passion for test and the impact it can have on product quality and the industry.

Weifeng Yao is a Software Development Engineer in Test at Microsoft Corporation. He joined Microsoft in 2008, and worked in Office Communications Group China before moving to Lync client team in Redmond, WA in 2009. He is passionate about exploring different test methodologies and push high quality product, and solving hard technical problems.

1. Introduction

Coined by Jeff Howe and Mark Robinson in the June 2006 issue of *Wired* magazine (Howe, 2006f), the term *crowdsourcing* describes a new web-based business model that harnesses the creative solutions of a distributed network of individuals through what amounts to an open call for proposals. Simply defined, crowdsourcing represents the act of a company or institution taking a function once performed by employees and outsourcing it to an undefined (and generally large) network of people in the form of an open call.

Testing of a product usually involves a large set of configurations on which each feature needs to be exercised. With every release of the product, legacy clients and features need support while new features get added. This further increases the cost an organization needs to pay for testing and stabilizing the quality of the product.

In an organization employees often use the product in development themselves (a practice called "dogfood") for a good amount of time trying to iron out bugs that need fixing. The channel to collect the list of issues is usually an email distribution group where the internal Microsoft users post their questions on the failing scenario. They rarely send out a periodic list of scenarios that succeeded for them.

In this paper, we describe how the organization can use the Scenario Compass tool to deploy, gather and analyze usage data, both success and failure scenarios, from the crowd without needing them to change any of their execution, how the test organization can focus their efforts based on execution of scenarios by the crowd. The premise is that if a large set of users are hitting a scenario, at least a handful of them would report any visual glitches which tools like Scenario Compass cannot identify. We also describe how the data collected by Scenario Compass is not limited only to the test team, and can be used by the User Experience team to identify the utility of features; the performance team in identifying real world performance numbers, the developers in predicting the amount of unit test needed on a feature they plan to change based on coverage it has from the crowd.

We begin our paper by explaining the problem space and the challenges faced due to a large test matrix. Next, we cover details of Scenario Compass, followed by the architectural design and results generated from the data collected from it. We then conclude the paper by presenting some key takeaways from this project and efforts that we are making for the next version of the tool.

2. What Is the Problem Space

The variations of tests to execute are huge and continue to grow from version to version. Each new release adds more complexity to the matrix. Just some of the variables in the current matrix are:

- The different OSs and architecture of those OSs
- The different architectures of the client product
- The install method of the product
- The SKU of the product
- The current version of the client to interop with, the previous version of the client, and the version before that.
- The current version of the client to be able to work with same and previous version of the server
- Interop with other legacy clients
- Different modes in which the client can be run.
- For the Lync client it could be desktop mode, remote desktop with Virtual Desktop Infrastructure mode on/off, Pairing with deskphone on/off
- Different versions of supported browsers
- DPI and screen resolutions.
- Touch & non-touch
- Topologies (OnPremise, Online)

Given the extensive test matrix, it's not a good ROI for the test team to execute all their tests on various combinations of the configurations. In such situations the test team executes the tests on some of the important configurations and can then rely on the masses to execute scenarios on several different configuration variations.

3. Why Create Scenario Compass

In a services environment, engineers can build logic into the product to direct and measure usage for the purposes of Testing in Production as well as Active and Passive monitoring. Telemetry and A/B testing is a key component in large scale production deployments. The architecture allows the administrators and engineering teams to measure, change, and react in real time to problems or respond to patterns observed in user choice.

A traditional desktop product often lacks the telemetry or mechanism to store, analyze, and react to real-time telemetry from the user. The desktop application may not have a server backend or the communications protocol may only contain business logic specific to the transaction at hand. Metadata may not be available or may not be of the fidelity desired for TiP (Testing in Production). In the examples above, install methods, architectures, browsers, DPI, etc. are client specific configurations which may not be in the protocol between the client & server.

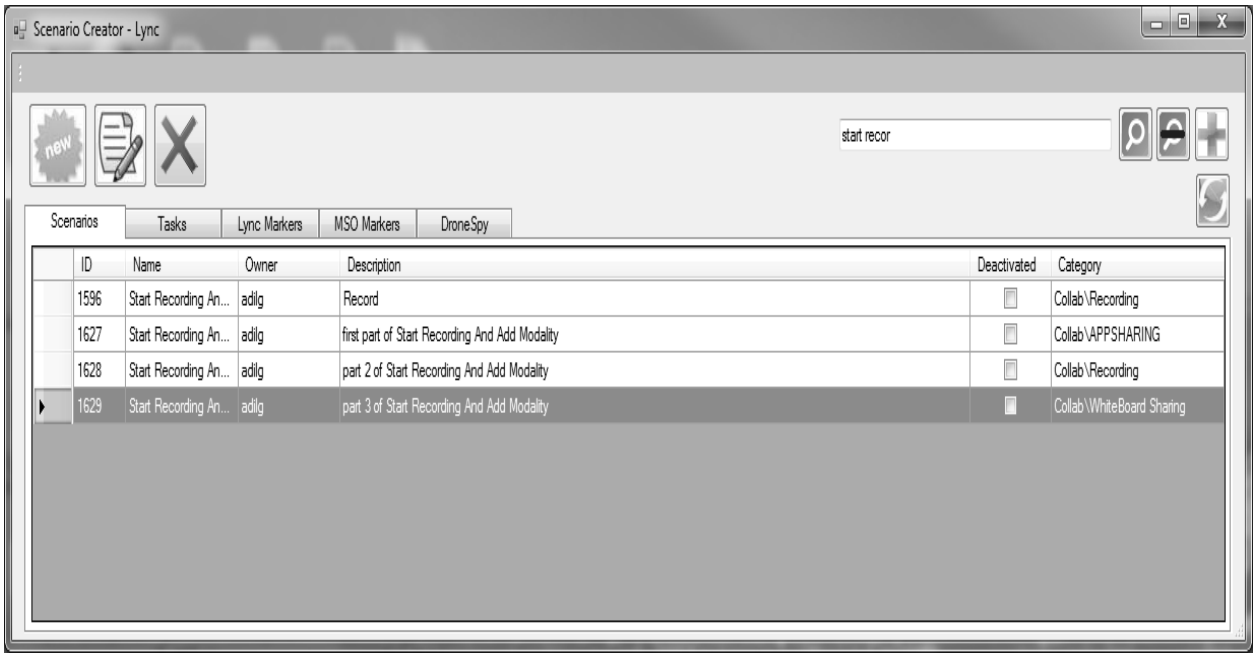
To fill this need, we built the Scenario Compass. Scenario Compass will “observe” the target product through a combination of Out of Process events, ETW markers, or proprietary APIs/performance markers. Actions can be monitored, collected, and analyzed for usage. Scenario Compass will gather the configuration data from the user’s machine. Scenario Compass will then upload the data to a central store where it can be aggregated & processed. The scenarios are associated with the configuration data for a complete picture of “what” happened and “how” it happened. Scenario Compass also provides a mechanism to send a response back to the client application to directly engage the user if desired.

4. What is Scenario Compass

Scenario Compass is a tool created by the Lync Client team and can be visualized as two separate applications:

4.1 Scenario Creator:

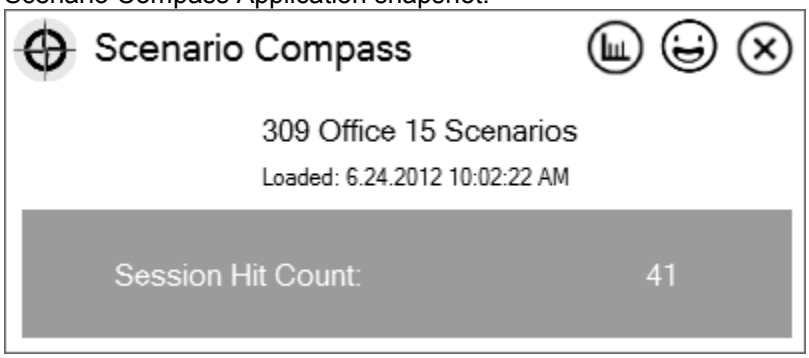
- The test owner needs to define which scenario is of interest by defining a set of events that get fired when that scenario is executed by someone.
- Using the Scenario Creator tool, the feature area owner defines the ordered sequence of events that make up a scenario.
- Once the scenario is defined, it is pushed up to the database and down to the Scenario Compass application.



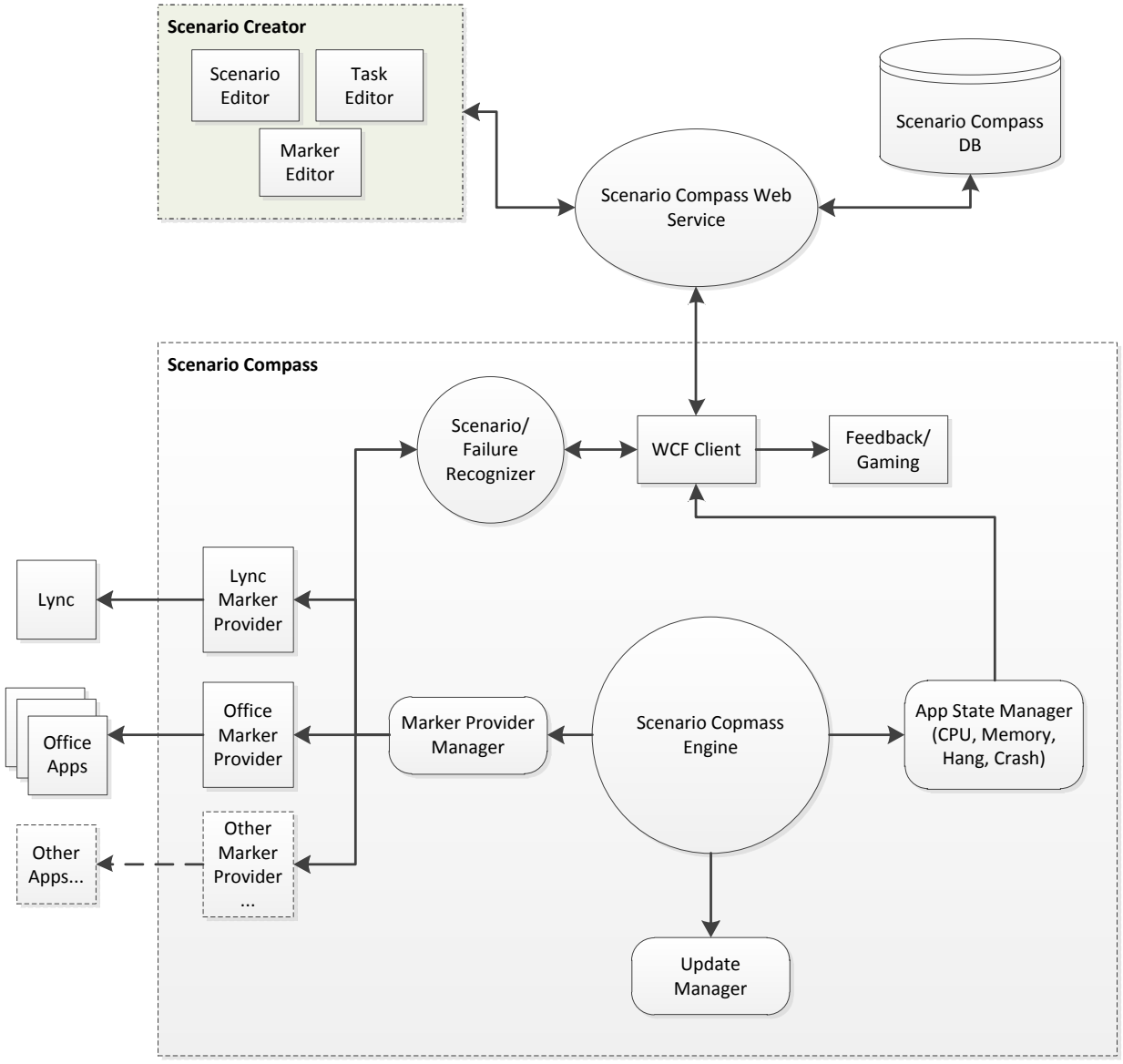
4.2 Scenario Compass Application:

- Scenario Compass is a lightweight application that listens to product instrumentation.
- Scenario Compass application is installed on the internal Microsoft users' machine.
- It attaches out of process to Lync and Office applications and listens for events fired by them.
- Based on the scenarios defined in the database, Scenario Compass application maintains an internal state machine to verify if any of those scenarios are hit and uploads the occurrence to the database.
- Scenario Compass also uploads various machine configuration information like OS, architecture, install type, etc.

Scenario Compass Application snapshot:



5. Design



Architecture Overview

Scenario Compass application is a Client-Server system. It has 3 core pieces including Scenario Compass client, the web service and storing Database.

As a high level overview, Scenario Compass client listens to the markers (data points in source code) from different marker sources, recognizes the markers as scenarios using definitions retrieved from the web server, and submits the hit data to the web service. The web service will process the data and store it in a database. Besides scenario hit data, Scenario Compass client also recognizes scenario failures and target application resource usages; for example, CPU and memory usage when target application exceeds a threshold.

Scenario Compass client is designed to be extensible to handle markers from different sources. We use the term Marker Provider for this role which is responsible for collecting raw marker data from marker sources and feed the scenario/failure recognition component.

Currently we have 2 types of marker providers - Office marker provider and Lync event provider. Office marker is a data structure instrumented in the office applications mainly for performance measurement purposes. When office applications perform different actions, corresponding marker data will be written to a shared memory space so Scenario Compass can read it. Lync event marker doesn't require additional instrumentation in the product. Instead, it utilizes the rich platform features and testability designs built in to the Lync application. More specifically, Lync client will start a COM interop layer when it starts. The interop layer allows out of process consumer applications to access (a.k.a interop) the properties and subscribe to the events. Scenario Compass is one of such out of process consumers. When Lync event marker provider sees Lync client starts, it will subscribe to the events and receive notifications from Lync client through this interop layer. When an event is received by the provider, it will be converted into marker that recognition component can understand, and recognize them as scenario hits or failures, which is the same process as Office markers.

The Scenario and failure recognition component is a finite state machine implementation. During Scenario Compass client startup, the state machines will be initialized from the scenario definitions created by Scenario Creator tool. At runtime, each marker will be tested by the state machines to determine if a marker will push the state machine to the next state, or a failure state. When a state machine reaches its end state, a scenario hit is generated and submitted to the web service.

For Scenario Compass client to work effectively, scenarios should be created prior to the data collection. This is done by Scenario Creator. It is a data centric UI application that allows feature owners to enter scenario definitions he or she want Scenario Compass client to recognize. A scenario can be considered as sequence of expected actions. In Scenario Compass, a scenario consists of a sequence of tasks, and a task consists of a list of markers. There are 2 ways to create a scenario, build bottom up or record and play. Using the first method, feature owners will search and create markers, then create tasks using the marks, finally create scenarios consist of the created tasks. Using the second method, record and play, feature owners will run Scenario Compass in a calibration mode to run the scenarios and collect marker sequence for that scenario using Scenario Creator, after the markers are collected, feature owners can select the markers they're interested in and save as tasks and scenarios. The second way makes the scenario creation much simpler.

6. Case Study – Lync Deployment:

A key part of our test portfolio is known as our “Dogfood program.” The term comes from dog food manufacturers having to taste their own dog food to deem whether it was desirable for consumption. In the context of Lync, we find that a large scale internal deployment of the preproduction application helps find issues that may not be found in local testing.

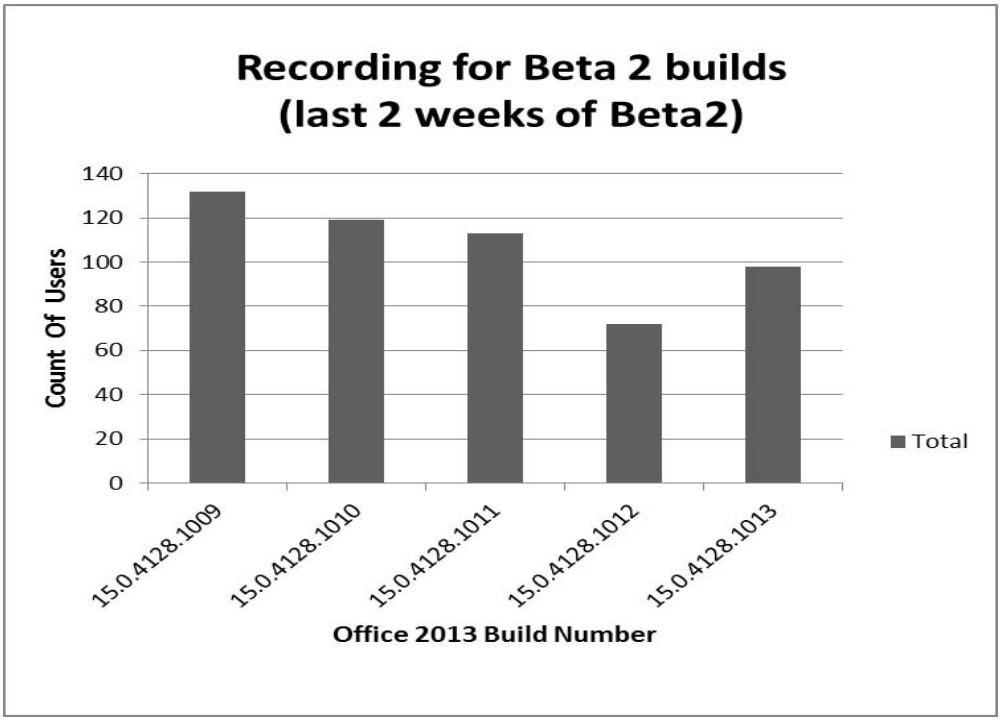
The dogfood program relies on a key crowdsource tenet – that at scale, users will complain & report if they encounter issues. The top issues will then reveal themselves and can be corrected before shipping the product. Conversely, one of the challenges of a Dogfood program is in detecting “what went right.” It may be hard to distinguish what was working properly versus a feature that may not be used heavily in the deployment. Similarly, even if a feature is used frequently, it is hard to determine the spectrum of configurations covered in the usage of that feature.

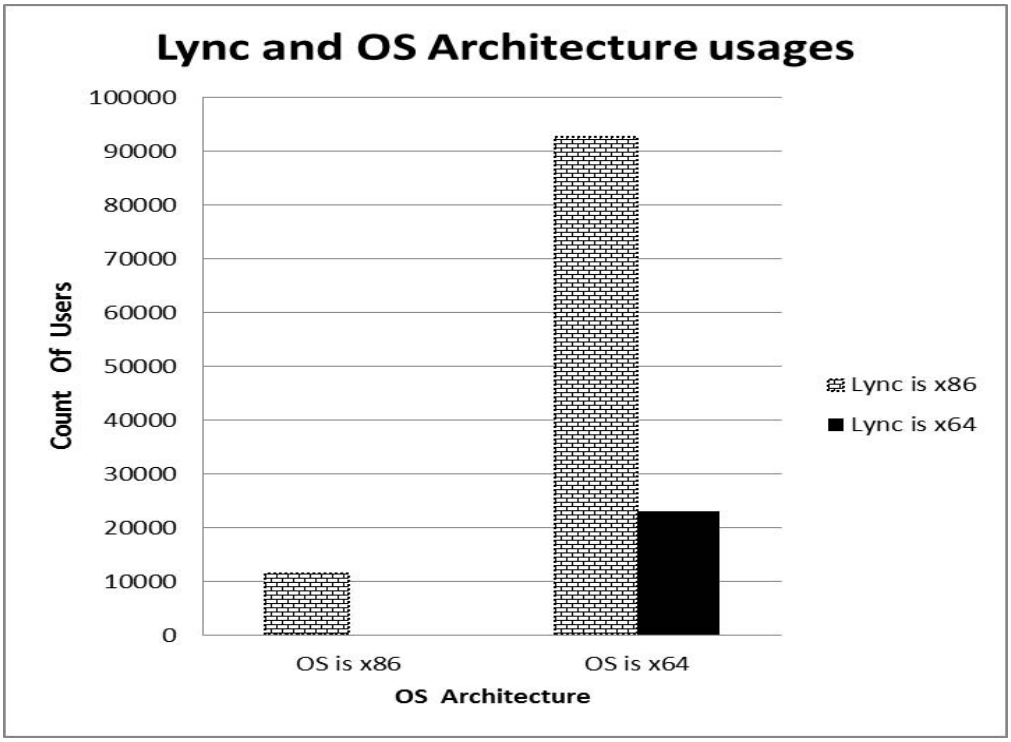
For the Lync Dogfood deployment, we deployed Scenario Compass alongside Lync to fill in this information. This required each test owner to define the scenarios in terms of events that the tester would want to measure.

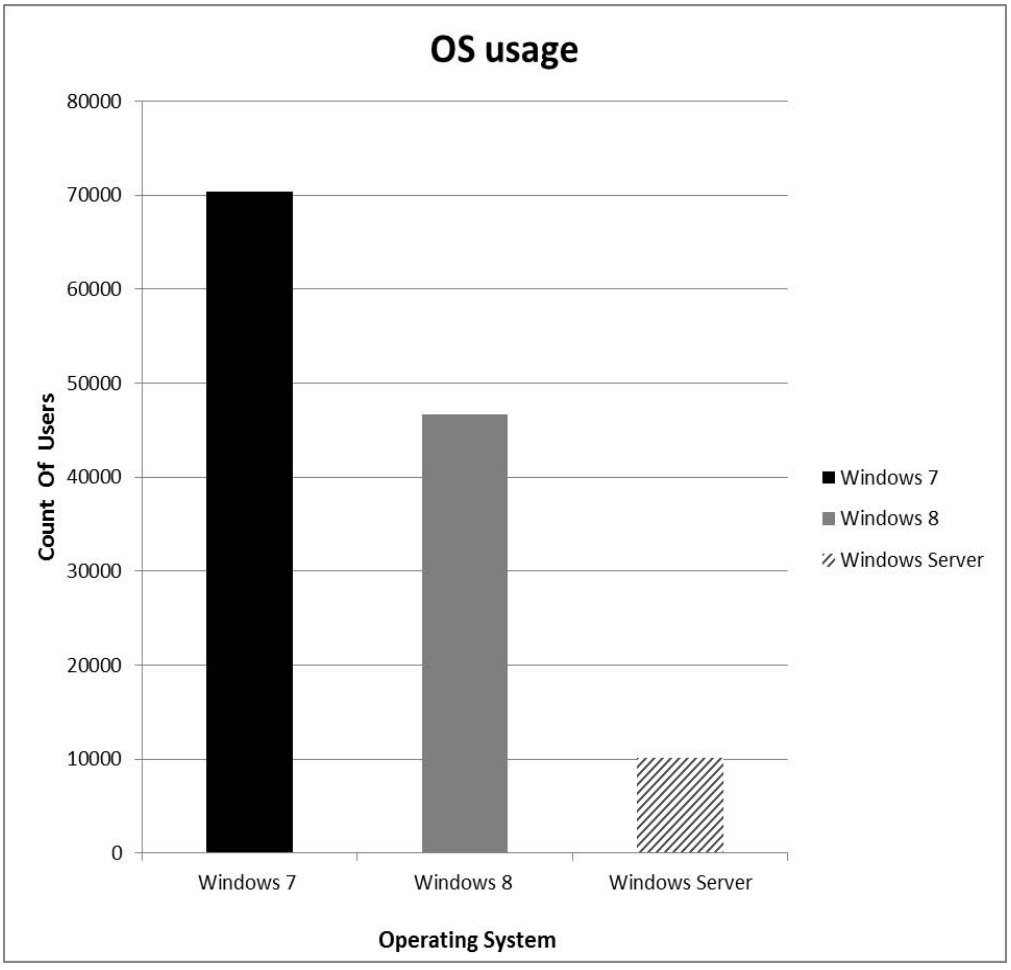
An area owner can decide to view results based on how many dogfooders hit his/her feature scenarios on a particular build or on a particular topology, for example how many users used the product in high DPI, if users used the feature on an x86, x64 architecture, and so on. Scenario Compass can provide all the data needed to quantitatively answer these questions.

The feature area owner can use our website to pivot the data off several variables. This chart shows the scenario coverage of Application Sharing feature are for the last 3 days on the last 3 Beta2 builds of Lync on specific topologies.

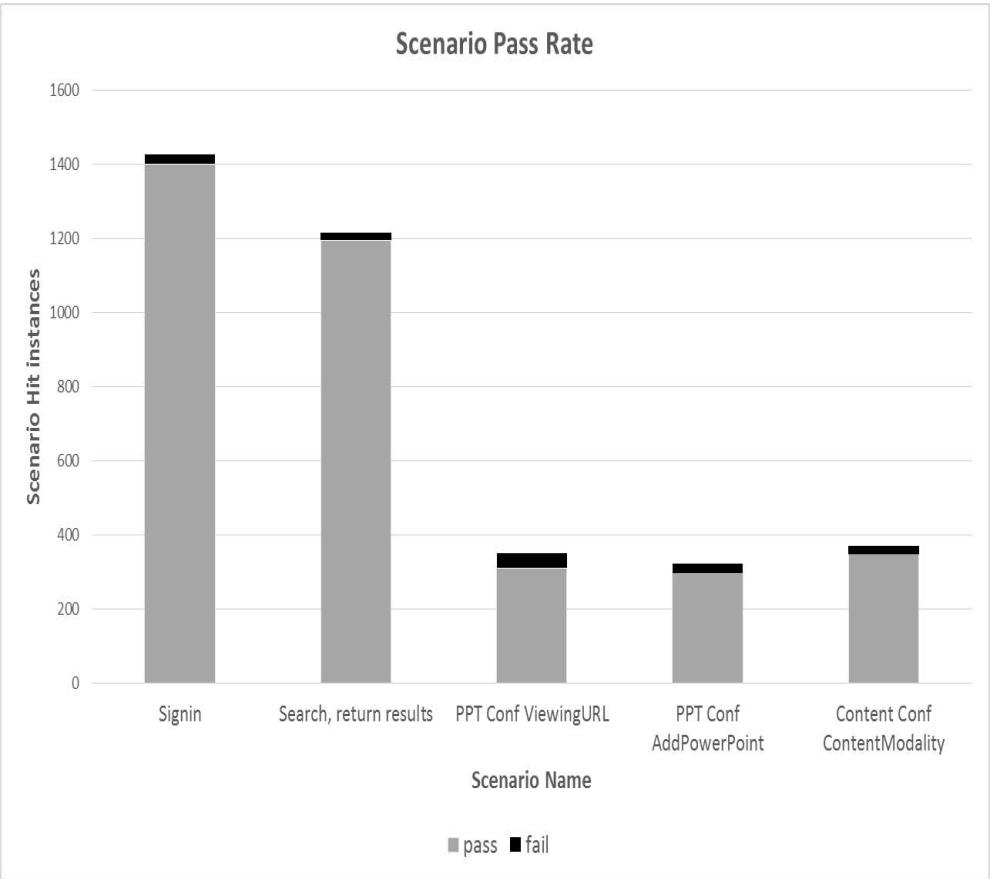
Scenario coverage by build.




Lync and OS Architecture usages
100000
90000
80000
70000
60000
50000
40000
30000
20000
10000
0
Count Of Users
OS is x86
OS is x64
OS Architecture
Lync is x86
Lync is x64


OS usage
80000
70000
60000
50000
40000
30000
20000
10000
0
Count Of Users
Windows 7
Windows 8
Windows Server
Operating System
Windows 7
Windows 8
Windows Server

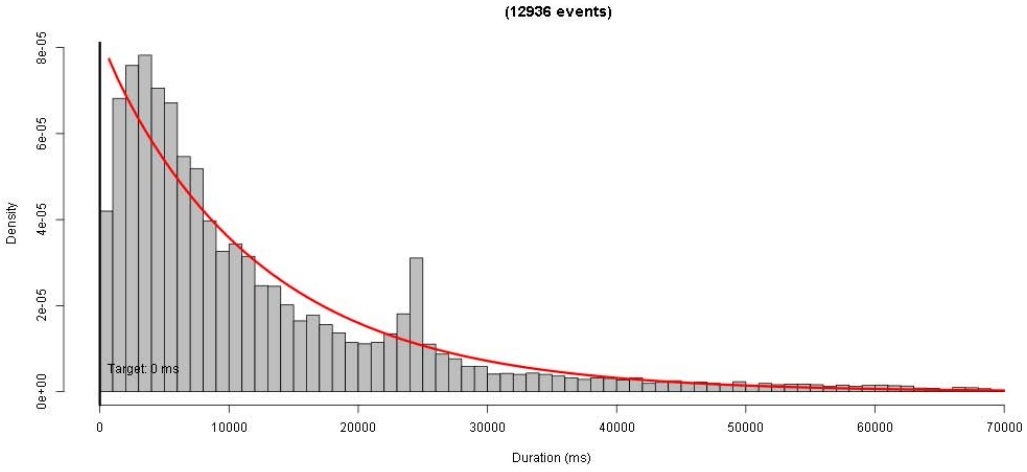
Scenario failure information:
An area owner can decide that his scenario should complete in a given amount of time when defining the scenario in Scenario Creator. The time taken for a particular scenario can be calculated from the data uploaded by Scenario Compass and based on that a scenario can be marked as pass or fail.
This timing information can also be used by the performance team to identify the how long their scenarios take to execute on the user's machine and compare the data (if any) with that taken in the lab.



Using target percentages per configuration, one can create a heat map as such of the test matrix (variations of tests), so the team knows where to focus efforts on.
The heat map below has been created using data collected for Beta2 and percentage of overall usage set for each cell. E.g. this heat map of OS usage, conveys that Office application usage on Win7 has high but usage on other OSs like Windows Server 2008 R2 and Windows 8 was low.

Scenario Type	OS	Install Type	OS architecture	Lync Architecture	Network connection
AV Conferencing	Windows 7	MSI	x86	x86	Wired
PPC	Windows Server 2008 R2	C2R	x64	x64	Wireless
MSO_Office	Windows 8 Consumer Preview			WoW64	
Audio	Windows Embedded Standard, Service Pack				
Mso					
Search					
DataCollab					

Based on data we can build performance measurement charts as below.



From this chart we can identify that there are several users that are way over the original performance target planned by the team. This helps in readjusting the expectation and picking a more realistic target and to investigate the users that are over the mark.

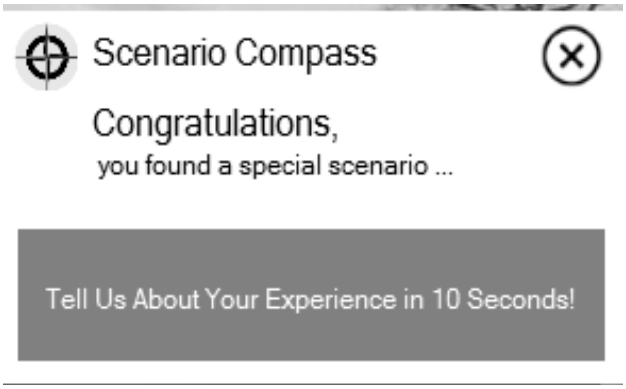
7. Interaction:

7.1 Feedback:

We built a channel in Scenario Compass to be able to reach the dogfooder. This is currently used to notify them that they have just completed a special scenario and we'd like their feedback on their experience of the scenario.

a. End of call rating and tying QoE data

We can pick any scenario defined in Scenario Creator and tag it as a 'game' scenario. We can also associate an action Scenario Compass would take when that game scenario is hit.
We have tagged end of a call as a game scenario. When the user hits this scenario, Scenario Compass, pops up a toast near the systray like this



When the user clicks on the toast we launch a survey that the user can take to describe his experience.



Once the survey is completed the response is recorded. Scenario Compass uses a context tag to correlate the survey response and the scenario hit information that it uploads. The QoE data that Lync uploads automatically can also be correlated to the scenario hit information, tying all the configuration information, the QoE data and the survey response together.

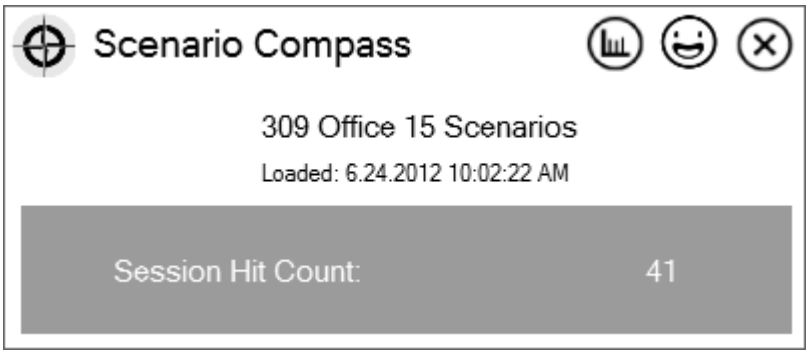
b. Recording playback by dogfooders.

Recording is a feature in Lync. The true E2E test for this feature is to record a call, conference and then play it back to ensure the recording completed successfully.

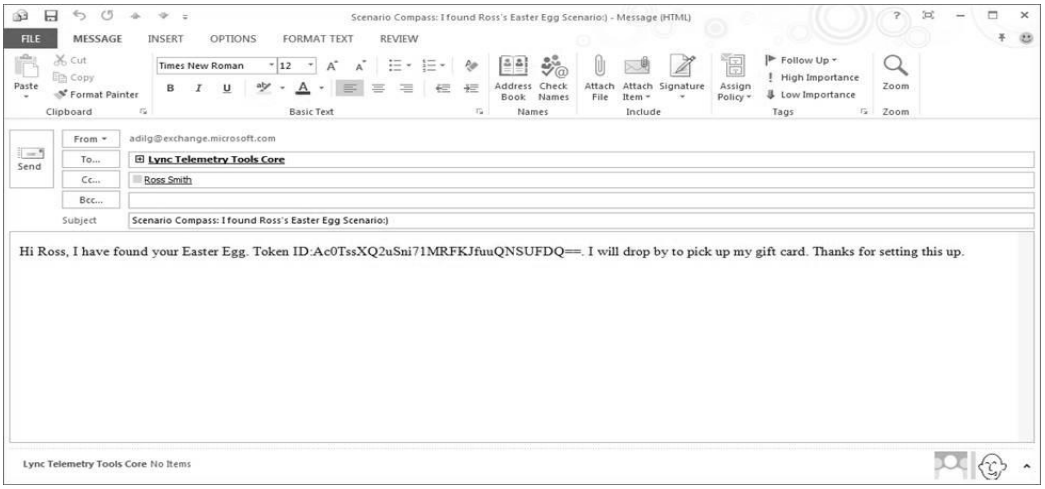
We are in the works to have another game in Scenario Compass where we pop up a toast to users that complete a recording session and request them to play back the recording and give feedback on its quality.

7.2 Incentivizing Users Through Games

- With Scenario Compass running on several user's machines we want to encourage people to execute certain scenarios or use certain features. We can show the scenario of the day on Scenario Compass's main UI and give an indicator of it being completed by the user or not.



- Easter Eggs
  Our Test Director has hidden an Easter egg Lync scenario in Scenario Compass which when hit by a user would pop up a toast which would lead to an email as below. He also provided several clues to help folks focus in certain areas to hit this Easter egg. We found that these incentives helped increase Lync feature usage of sparsely used features.



8. Key Takeaways and Next Steps

This project has been enlightening on several levels. Architecting and building the tool for scale presented the majority of the technical challenges. Running the dogfood program with sufficient participation to collect the data was a separate set of challenges.

We found three main tenets that are key to success:

1. **Good scenario definitions is the key**
   Identifying which scenarios will provide the best ROI (in terms of reducing manual tests) is very important. The right scenarios help confirm that a particular configuration is covered and hence a subset of features can be skipped. These valuable results that help the feature team can only be achieved if the quality of the scenarios themselves is high. To make Scenario creation easier, we created "Scenario Spy" to help capture the correct events from Lync.

2. **Large set of users is important**
   Having a large set of dogfooders is important to get more use of your scenarios. You can get automatic triage of sub features based on its usage from dogfooders. For example, towards the end of a milestone if a bug fix needs to be made in an area that has very shallow coverage from a large set of users, then the feature team has the usage data of the feature which could help in determining whether the fix should be taken or not. Results from the usage can also provide a direction on which scenarios are more frequently used, so more feature weight can be added there. Knowing which scenarios are failing help the owners to get a better sense of product quality and where more testing is required. Having a large scale deployment is needed for the data to be statistically significant to make these decisions.

3. **Built for extensibility**
   Historically in every release we add additional backwards compatibility, additional support in the product, and more integration points. The test matrix continues to grow whether it's configurations, combinations, etc.. With the appropriate investments in Scenario Compass scenarios and config capturing logic, the feature team (with roughly the same number of resources) can handle the additional test burden. In addition to measurements, the tool adds a new channel for owners to engage with the users to help cover the increasing number of scenarios. Using 'Games', a feature area owner now has a medium to get that feature more visibility. Performance data for features can now be collected from the users which are more indicative of what the customers will experience rather than lab run results. Easter egg games are a big incentive for people to try out different scenarios.

There are also some more subtle optimizations we learned over time in terms of the deployment.

a. **Less distraction for users**
   Users are very sensitive to their attention in the world of ever-increasing distraction. To minimize the "distraction footprint," Scenario Compass runs in an extremely quiet mode. We scaled the UI down as the deployment scaled up. When the user base of Scenario Compass was small we were able to have a higher touch engagement. If that meant prompting the user to click the 'ok' button to repair then we would pop up a UI for that. With the ramp up of the number of users we had to shift the attention to not being in their face at all. Once we had statistically significant deployment numbers, we opted for loss of data as opposed to bothering the users to repair for getting higher rates of data.
b. **Having a bank of machines**
   Having a large set of dogfooders means a significant increase in traffic to the web service and database. We had to move to a model where the web service, database and the update/install location for the tool were all on different physical machines. A 'Fail Safe' shutdown switch was added – this is a mechanism to automatically shutdown Scenario Compass in case we identified an issue after deployment. We implemented not one but two Fail Safe switches for Scenario Compass. When either of them are enabled Scenario Compass shuts down. One of the switches is a file on a file share and the other is an http link in case the file share is not accessible. We have used the Fail Safe switch a couple of times to make some fixes that came in as design change requests from management. This helped us achieve our "less distraction" goal for the end users.
c. Resilient start of Scenario Compass: We want Scenario Compass to be running any time the user is logged on to the machine. For this we created scheduled tasks to launch Scenario Compass on log on and at midnight each night. The reason for midnight SC was to start SC in case the user did not log off and back in and Scenario Compass was not already running on the machine (say Fail Safe Switch was now disabled).
d. Batching and throttling of upload information:
   With increased users, and even after having separate machines for the web service and database, the traffic to these components becomes unmanageable due to limitation of network, number of requests the web service can handle, and/or the CPU of the machine. We had to update our design to include batching of the scenario hit information to the web service in intervals of every 5 minutes or when 15 scenarios are hit, whichever is earlier. We also had to include throttling of the access to the WS or update location. Earlier, each Scenario Compass

would query the WS for a large amount of metadata. For example whether a new Scenario Compass build was available for auto-update, if the Fail Safe switch on, are new scenarios available, etc. This caused a flood of requests to the WS which could not handle the load. We accomplished throttling by randomizing the start time of each Scenario Compass when the user logged in or after midnight, upto a max of 5 minutes delay. This randomized the polling time for each Scenario Compass since they all were relative to Scenario Compass's start time.

Some of the next steps, to increase the versatility of Scenario Compass that we are investigating are:

- Automatic uploading of logs when a failure scenario is hit. This will help in debugging the issue in addition to knowing the frequency of failure.
- With increased number of users, Scenario Compass itself should be A/B tested as changes to Scenario Compass, the games, or the scenarios are rolled out.
- Having the ability to configure which games a particular user could participate in. We can decide that we don't want to distract senior executives with all the games the rest of the users are participating in until they are ready.
- Feature of the day. We are putting the final touches on being able to influence the users to try out certain scenarios.

9. Conclusion

We found that having a client side Telemetry application has been a significant investment that is paying dividends. Scenario Compass has provided us a large amount of data that has guided us to make business decisions around where we spend our test resources. It's filled a gap that's existed in desktop applications for some time.

Like all solutions, Scenario Compass has its pros & cons. Being aware of these pros/cons, mitigating or addressing the cons, and making the investments is a business decision that everyone has to make.

- Some of the Pros:
    - Detection of configurations
    - Detection of scenarios encountered cross-referencing the configurations in which they were measured
    - A/B testing is now possible for desktop applications
    - An avenue for direct user engagement allowed us to incentivize certain behaviors through games and rewards

- Some of the Cons:
    - Scenarios have to be explicitly defined in order to be measured. Quality of the Scenario definitions played a key role in usefulness of the data.
    - Usefulness of the Scenario Compass data is proportional to the scale of the Dogfood deployment – it requires a good dogfood program deployment.
    - Need to pay the costs associated with maintaining Scenario Compass & related infrastructure

For the Lync team, we moved ahead with the investment and it has paid off in helping us ship our next Wave of products.

Acknowledgements

Thanks to those who worked on this project. Mahesh Ambravaneswaran, Bruno Torres , John Gilbert, Purvi Vaidya , Vijay Jayaraman, Shankar Arigela , Veronika Ruzin , Joshua Williams, Robert Musson, Harry Emil, Andy Lim, Ross Smith.

The Power of Process, People, and Tools When Testing a Complex Integration Landscape for a Very Large Initial Retail ERP Implementation

Nicki Nicolo
Nike, Inc.
Nicki.Nicolo@nike.com

Abstract

Anyone who has ever led the integration testing effort for a large project with a complex integration landscape has had a difficult time finding guidance on what is necessary to succeed in this type of enterprise. There is scant information available that provides any meaningful assistance. This paper presents a real life project as a case study on how the nexus of process, people, and tools is the foundation for enabling a high quality delivery.

The Nike Retail Enterprise Resource Planning (ERP) team created a number of tools including complex data and transaction trackers to allow communication across partner systems and business functional tracks. Additionally, there was a well-planned test schedule. The Retail ERP team utilized existing off the shelf tools such as Quality Center for test requirements, test cases, test execution, and defect management.

The implementation for this case study involved the initial implementation of a Retail ERP system with 26 partner systems, approximately 150 interfaces, 5 integration tools, and systems belonging to other corporations.

The paper will close with a look at ideas for improvement based on the issues the Retail ERP team faced and what the Retail ERP team learned along the way.

Biography

Nicki Nicolo is a lead software QA engineer at Nike, Inc. in Beaverton, OR. Nicki has worked at Nike for 26 years starting out as an MVS systems programmer and also working as a project manager for 6 years. She has 33 years of experience working with both software and hardware. Nicki has a Bachelor of Arts degree from Mount Holyoke College.

Copyright Nicki Nicolo 2012

1. Introduction

This paper is a case study of integration testing, from soup to nuts, for a "greenfield" Merchandise (Merch) Retail Enterprise Resource Plannning (ERP) implementation at Nike, Inc. We will explore the nexus of process, people, and tools as the foundations for enabling a high quality delivery of the integration portion of this implementation. The paper will close with a look at ideas for improvement based on the issues the Merch Retail ERP team faced and other key learnings along the way.

The main functions of this Merch Retail ERP system are financial and product planning for stores, procurement and distribution of product for stores, sales and other operational reporting for stores, inventory management for stores, product movement between stores, pricing changes for product in stores, nightly financial reconcilliation processing, processing invoicing and product returns with vendors, and providing general ledger information to Nike's financial system of record.

When I was asked to take the integration Quality Assurance (QA) lead positon of the Retail ERP implementation, I spent some time in research looking for information on how to test integration in a large and complex environment. I was looking for both test strategies and case studies that would reveal the types of issues our team might face. I found nothing. I also attended a flash session at the Pacific Northwest Software Quality Conference (PNSQC) that year and asked for assistance. People talked to me about the issues I could expect to face. However, no one pointed me to any literature in this area. It's my hope that this paper will be the beginning of a greater body of work in this area.

2. Background

The project for this case study involved the initial implementation of a Retail ERP system with 26 integration partner systems, approximately 150 interfaces, five integration/middleware tools, and some partner systems belonging to third party vendors. The Retail ERP system was designed and constructed along the lines of eight business functional tracks (Master Data, Finance, Purchasing and Distribution, Inventory Management, Sales Analysis, Financial Planning, Assortment Planning, and Reporting). Testing was structured to follow the business processes under each of those tracks.

Eight of the integration partner systems were considered core partners. The core partners are other Nike systems such as global store systems (brick and mortar), digital stores, our wholesale ERP system, our Warehouse Management Systems (WMS), digital store WMS, our sales tax system, and our markdown optimization system. Retail ERP test efforts needed to be tightly coupled with those of the core partner systems because any severe issues with these integration points could result in major business impacts.

The program management team determined that there would be four testing cycles for the entire implementation. Quality Assurance Streamlined Test (QAST) was the early testing phase and was scheduled to last five weeks. The purpose of QAST was to ensure that the system and interfaces were ready for end-to-end testing. The integration test effort during QAST utilized grey box techniques to validate the interfaces functioned according to specifications. System Integration Test (SIT) 1 and SIT 2 were the two end-to-end testing cycles. Again, testing was based on grey box techniques. SIT 1 was scheduled for six weeks, while SIT 2 was scheduled for seven weeks. SIT 2 actually continued through User Acceptance Test (UAT) due to late code deliveries and major defects that needed to be resolved. UAT was the user acceptance cycle and lasted for three weeks. Our focus was to ensure the middleware transformations provided the proper mapping of data fields between integration points and created any derived data correctly. We also validated error handling.

At Nike, unit testing is performed by the application engineering teams prior to the code being delivered to the formal QA test environments.

The QA and implementation teams were comprised of onsite and offshore consultants , and Nike employees. A Nike employee led the effort for integration testing and created the strategy and plan for integration testing. The functional track leads were consultants and were responsible to make sure that the Merch Retail ERP tests were designed and executed correctly for their respective tracks. Our test partners in other organizations were responsible for determining and executing any additional testing they deemed necessary. They were also responsible for determining the impact of defects to their applications.

Since the majority of this implementation's integration relied on batch processing, we combined the test efforts for integration and batch. The batch system consists of scripts that trigger the running of various processes. The batch system holds the scripts and determines when they should be executed, either based on a schedule (day and time) or some other trigger such as the completion of another batch job that is a predecessor.

Combining integration and batch testing allowed us to work efficiently. However, this meant that the development teams had to tightly coordinate the delivery of interfaces and the batch jobs to run those interfaces. The test team was trained on the use of the batch tool so they could run all jobs independently. In general test teams at Nike must request the batch team to run the jobs for testing. However, testing is not the batch team's priority.

3. The Power of Process, People, and Tools

Here are the major points revolving around people, processes, and tools that were key to the success of the Nike Retail ERP QA team's efforts:

- We implemented a single point of contact process that ensured cooperation and accountability with our partner systems, both for testing and for defect remediation. It also helped because some of the teams involved are very large and project members outside of that team may not know who to contact.
- We tested the interfaces as early as possible so that we had a solid platform for end-to-end business process testing. Initial integration testing used black box techniques to find transformation mapping errors.
- We created a test data tracker that allowed us to have clear visibility to test data as it crossed system boundaries and also ensured that testers were not stepping on each other as test data was coordinated in the tracker.
- We managed all code promotion tightly so we were testing on known platforms.
- We had a shared test calendar so test activities were well coordinated and well known across all of the teams involved.
- We had a dedicated team room for formal and informal meetings and triage work with representatives from all the involved teams.
- We had a well run defect management process. This allowed us to triage quickly and prioritize fixes.
- We coordinated closely with our key partner system test teams via the daily stand-up meetings where participation was mandatory. This allowed us to change and discuss daily plans, outstanding issues, and key fixes that needed to be tested.
- We clearly defined test and defect responsibilities across the various teams so there were no grey areas.

4. People and Roles

Early on in the project we utilized the ***Single Point of Contact (SPOC)*** concept with our partner system test teams and partner system development teams. This was useful as some of those organizations are very large and our team had limited knowledge of who we needed to contact for various tasks and activities. As the project progressed we lowered the level of SPOC touch points to the various system functional levels. This allowed us to operate efficiently, in the face of complexity that would have otherwise been very difficult to overcome.

Our project team was trained on our processes and tools. The team also gave frequent feedback that led to ongoing improvements in our processes and tools

Merch Retail QA Manager - Overall responsibility for Merch Retail Quality Assurance.

Merch Retail QA Project Manager – Maintain QA project plan and facilitate resolution of issues and mitigation of risks.

Merch Retail QA Test Manager – Manage and track the execution calendars and provide weekly status reporting.

Merch Retail QA Leads (Functional Lead, Integration/Batch Lead, and Reporting Lead) - Manage the execution of tests, coordinate with partner system test teams, provide inputs to Weekly Status reports, verify test results and defects, manage issue resolution and follow up with partner system QA, and track daily progress and raise issues.

Defect Triage and QA Build Mgmt Leads - Schedule and lead Defect Triage Meetings and approve transports/builds prior to move to QA environments to ensure a stable test platform.

Partner System QA Leads – Manage the execution of partner testing, coordinate with the Merch QA Integration Lead, and provide Weekly Status reports to Merch QA Integration Lead. Some of Nike's technical teams do not have separate QA roles so this role was taken by a business system analyst with testing responsibilities for that team.

Testers - Execute test cases, record results, report defects, and validate fixes. Some of Nike's technical teams do not have separate QA roles so this role was taken by a business system analyst with testing responsibilities for that team.

Business Systems Analysts – Consults with QA Leads and participates in defect triage meetings. These team members can clarify issues/defects and drive their resolution with the AE teams.

Subject Matter Experts – Business team members who consult with QA Test Leads and participate in defect triage meetings. These team members provide the business perspective as to the criticality of defects.

Application Engineers (Batch, Environment, Integration) – Consult with QA Testers, participate in defect triage meetings, fix defects, and prepare for code migration. Their primary job is to fix the defects and ask questions if they don't understand the defects.

5. Tools

Given the complexity of this project and the large number of people who worked on the project from a number of teams, most notably Merchandising Retail, Wholesale, Global Store Systems, Digital,

Warehousing systems, Markdown Optimization systems, and the Retail tax system, our toolset enabled both our people and our processes to help deliver a successful project.

QA Contact List

This list resided on the program Sharepoint site. The contact list names spanned all Nike technical teams across departments and all external companies involved in the integration testing effort so anyone on the QA team could easily find the correct person and contact information.

Data Tracker

This was a tool the Merch Retail QA team developed to allow us to track test data and transactions as they passed through all of the systems involved in integration. When I was seeking insights on what major issues this kind of integration testing would face, test data management was by far the most common point.

Given that data attributes are referred to differently by different Nike teams from different departments and business areas because of system differences, this allowed us to speak a common language and to ensure transformations were functioning properly. This tool started as a very simple spreadsheet to track product data as it flowed through various systems. However, the team refined the tool and morphed it into a very complex and complete tool. This provided great details during root cause analysis for defect resolution.

The Data Tracker was used to track fourteen critical data areas ranging from master data for retail stores, warehouses, finance, and product, to specific inventory to be used for end-to-end testing over all the wholesale warehouses and vendors to order transactions, to track orders from placement to receipt to invoicing and exception situations.

The Data Tracker was designed and built around the data flowing through the various business systems involved in the overall integration. Some of the systems involved were created by commercial software vendors. Others were created by Nike application engineering teams. And another group of systems was created by other external businesses. The interfaces involved were either created by Nike's middleware engineering team or in a few cases used the vendor provided functionality of certain systems.

Test Management

The team used an off the shelf software package to create test scripts, track test execution, and manage defect resolution. We trained all of the project teams on how to use this tool and created specialized views of defects relevant to various development and business analyst groups. This tool enabled our defect management process and our metrics and statusing processes.

Test Calendar

The Merch Retail QA team manually created calendars showing detailed, daily test execution plans for each test phase (QAST, SIT 1, and SIT 2). This tool helped all the test teams stay coordinated and let us understand how pre-requisite testing factored into the overall schedule. After the initial release, we enhanced the test calendar creation so that it can be created using the planned end and start date fields for the tests in the test management tool.

Batch Execution

The Merch Retail QA team was trained so they could manually execute batch jobs using the batch management system. This enabled the test team to speed up test execution as they did not have to wait on hand-offs to and from another team. There were also periods when the batch management system ran on the automated schedule to ensure the schedule was set-up correctly. Normally, the batch support team must run batch for testing. However, production support is batch support's first priority.

QA Team Room

The Merch Retail QA team had the use of a large, private, conference room which was used for both formal and informal meetings. Defect triage meetings and stand-up meetings were held in the room. It was also available for SWAT teams comprised of application engineers, business systems analysts, and QA engineers from multiple teams working to understand defects and find solutions quickly. Public conference rooms are often not available on short notice so that aided our efforts to reduce resolution time.

6. Processes

These key processes enabled the project team members to deliver on quality and keep the entire team apprised on the state of the project in terms of quality. For a number of reasons, QA processes within the technology teams at Nike are not standardized. Because this project spanned a number of Nike technology teams clearly defining, communicating, and utilizing this consistent set of processes helped everyone work together.

Design Reviews

The Merch Retail QA team was involved in review meetings for the integration high level design documents, the functional specifications, and the technical specifications. This provided invaluable assistance in providing the critical information for creating our test cases as well as ensuring that those documents provided clearly testable functionality.

Test Execution Management

We used our test execution management process to coordinate testing across teams and to provide daily metrics reports. We also asked our partner system test teams to provide similar reports. By and large, integration test management was done as part of the larger testing effort. We did track batch testing separately to ensure each job was tested at least once before the implementation went live.

Defect Management

We had a robust defect management process and we asked our test partners to report any critical defects in our system as well as using whatever process that team normally used. We kept a whiteboard of all very high and high defects that needed to be resolved in order to move into the next phase of testing or finally into business implementation. We made sure that the integration project manager was given immediate visibility to any defects that needed a quick resolution.

Code Promotion Management

We had a code promotion management process to ensure that code deployments and batch changes were done in a controlled manner on our test systems. Nothing could be moved without approval from a QA Lead. This ensured that our code base was always at a known state and alerted us when defect fixes needed to be tested. We also utilized soft and hard code freezes to provide a stable code base for “go-live”.

Daily Integration Stand-up Meeting

The Merch Retail Integration/Batch QA Lead ran a daily stand-up meeting for the core partner systems so everyone could be on the same page about what testing would occur on a daily basis and any issues could be discussed and escalated as necessary. This allowed all core partner systems to be aware of any issues with data conversion and system functionality.

7. Software Quality Strategy – Objectives, Activities, and Phases

Integration Test Objectives

- Ensure all necessary configurations and platform set-ups for integration functionality are correct.
- Ensure the outputs from the Merch Retail ERP integration links are compatible with partner systems.
- Ensure all data transformations meet business, technical, and legal requirements. Data must exit the source application/system in the expected format and arrive at the target application in the expected format and load into the target system correctly.
- Validate set-up of batch jobs.
- Validate batch scheduling set-up.
- Validate batch security set-up for batch logins/users for:
  - Retail ERP
  - Partner systems
  - FTP drop boxes

Planning Phase – Test scenarios

The functional track test teams created the signed-off test scenarios based on Merch Retail business processes and variations of these processes. These test scenarios were signed off by the Business System Analyst, Subject Matter Expert (SME), and Partner Systems QA. The test process scenarios were used to build the interface, as well as functional (system level), test cases for the project.

Development Phase – Test Case, Test Set Creation, and Interface Test Environment Validation

The QA team wrote test cases based on test scenarios, referring to the Business Process Documents and Functional and Technical Specification documents. Each application testing team (Global Store Systems, Markdown Optimization, wholesale, etc.) was responsible for identifying, gathering, and preparing test cases and test sets for their respective areas. Merch Retail QA worked with our partner QA teams to identify the integration points.

After all test cases for an integration development phase were created, test sets were created.

The Business Systems Analysts reviewed the test cases and provided the final sign-off for the test sets.

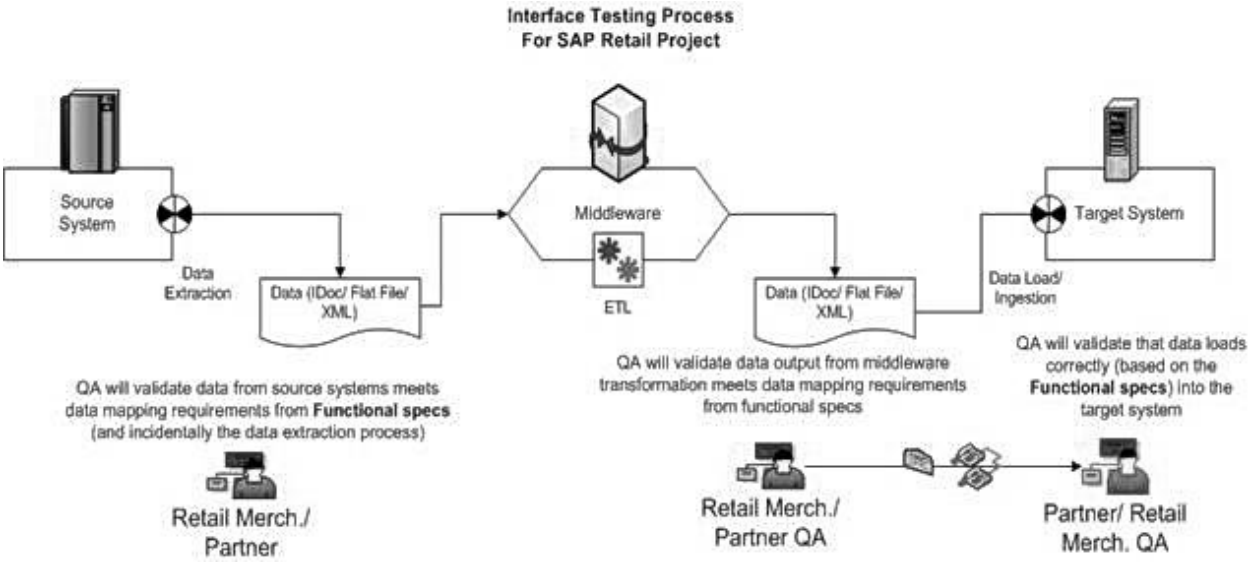
The Application Engineers specified input/output and unit test scenarios in unit test scripts which were a part of each functional technical design document. Unit tests only tested the functionalities/requirements specified in the business process and functional specifications. All unit tests were stored in the appropriate repositories.

QA team did not participate in unit testing. However QA team did refer to the unit test cases to refine test cases (if necessary).

Execution Phase

Retail Interface Testing – Early Test Phase (QAST)

All of the interfaces were new. Our chief concern early on was the accuracy of any transformations. We decided to use grey box testing for this phase as it met our needs. The goal of QAST was to enter SIT knowing that the crticial interfaces were functioning properly prior to starting end-to-end testing.



Objectives

- Validate that business critical integration functioned according to business and technical specifications prior to the start of full systems integration testing to allow SIT testing to focus on end-to-end business process testing.
- Test integrations, including connectivity, between Retail ERP components and the partner systems that are immediately adjacent to Retail ERP.
- Validate scripts and security for batch jobs was set-up properly.

Retail Interface Testing – System Integration Testing Phases (1 and 2)

Objectives

- Ensure all business integration was functioning as expected in support of end-to-end business process testing.
- Greybox level testing.
- Validate scripts and security for batch jobs was set-up properly for untested jobs from early phase.
- Validate batch scheduling via turning on the automated scheduler functionality for a period that simulates week in the life.

8. Overall Results of Nike Retail ERS Integration Testing

- Our metrics techniques for QAST did not allow us to separate integration testing from other testing done during that phase. Metrics became more important and were refined during the SIT test phases.
- 923 integration test cases were executed during the SIT 1 test phase. These covered batch testing as well.
- 818 integration test cases were executed during the SIT 2 test phase. These covered batch testing as well.
- Most of the integration and batch issues detected post “go-live” were minor defects or were, in fact, requirements changes.

9. Issues Encountered During Integration Test Execution

- One of our critical issues was late code delivery. The last code delivery occurred during the SIT 2 Test Phase. This did not leave us with much time to get all of our testing accomplished.
- There were issues with core ERP functionality that led most integration testing to be blocked for a significant period. The team faced significant issues with product-related functionality until late into the SIT 2 Test Phase. Again this led us into a compressed schedule. The daily stand-up meetings were invaluable for allowing us to understand what testing to prioritize based on what code and what fixes were available to us as well as the business priorities of the partner systems in terms of various scenario executions.
- There were data quality issues with converted data that forced the team to used mocked up data for testing. This was not optimal as we were not able to encounter data-related issues that existed in the system. This led to a few defects not being discovered until post “go-live”.
- There was test resource turnover with some of the core partner systems that meant we had to go through repeated ramp-ups with those teams and created the risk that regression testing that might be required by those systems would not be identified and executed.
- The test team faced issues with the availablity of core partner test system environments as they were not always in sync with Retail needs. Again we had to be flexible and nimble with our execution scheduling.
- The test team sometimes faced slow turn around of test tasks with some partner test teams. We did our best to work on relationships with those teams and escalate if necessary.

- The test environment ended up being shared by the training team and business analyst teams working on issues. This meant that QA faced testing in an uncontrolled environment. As much as we could, we tried to keep the test data separate to ensure that we understood which work caused the issues and to ensure that test data was not overlaid by teams who were not performing work for the test track.

10. Ideas for Improvement

- Data proved to be one of our biggest issues throughout this exercise. The systems involved are highly sensitive to different data permutations. Since this project finished, I have been exploring the use of combinatorial testing to provide deeper coverage in the areas affected by data sensitivity.
- Do not succumb to the temptation to test with mocked-up data because eventually the issues masked by using mocked-up data will surface. It's far better to find these issues earlier rather than later.
- Implement better change management processes for test scenarios and test scripts. This was difficult given the need to balance getting testing completed within tight windows and ensuring changes were reflected. We implemented a clean-up effort after the implementation went live so that our test scripts would be usable in the future.
- There was one major issue with performance in one set of interfaces post "go-live". In the future, integration should be more tightly coupled with the performance test track.
- When we started testing we did not understand how order processing testing worked with Nike's wholesale and warehouse management teams. We badly underestimated the time it would take for orders to be placed and fulfilled by those teams. In the future, we should engage our test partners to understand their test processes better.

11. Acknowledgements

I would like to acknowledge the Retail ERP test manager and Retail ERP functional test leads who took my concept of a test data tracker and created a tool beyond anything I initially imagined. This tool was innovative and strategic.

I'd also like to acknowledge the leads and project managers from our core partner systems for their time and efforts above and beyond, and their cooperation to ensure the success of this portion of the project. Without them, our final results would never have been this good.

The Black Swan of Software Testing – An experience report on exploratory testing

Shaham Yusuf & Venkat Moncompu
shyusuf@deloitte.com, venkatams@yahoo.com

Abstract: In an industry where prescriptive practice focused on requirements-based testing is prevalent, much attention is paid to scripted test development and verification for conformance. Some people take extra effort to provide visibility of test coverage to leadership that is sufficiently satisfactory yet products/applications continue to fall short of meeting the 'fitness of use' criteria of Quality. One reason is the dysfunctional relationship between a design that is system-centric and a solution that ought to be user-centric, which leads to user acceptance issues during software development or defect leakage to production.

Exploratory testing relying on continuous learning, test adaptation and execution, is an effective complementary testing approach which addresses above mentioned gap and helps to detect business critical defects quickly in the testing cycle. However, traditionalists revile at the suggestion of using exploratory testing citing reasons of coverage, test design, traceability, accountability etc. The curious *gen Y* testers are faced with the daunting task of winning over the skeptics to adopt the practice.

In this presentation we are sharing key factors that have helped us succeed in our efforts to convince stakeholders and to adopt exploratory testing in large scale independent testing engagements. We intend to showcase our blended approach (in other words, Alternative Test Design Techniques) where test designing was done on high level leveraging Scenarios, Mind Maps, Checklist, and Cheat Sheets. Also, the defect metrics presented would intrigue you to give exploratory testing another chance!

Biography:

Shaham Yusuf is a passionate tester and vivid learner and practitioner of software testing from Deloitte, Mumbai. He is self-inspired and believes in continuous learning and practice. Testing an application with an exploratory approach makes more sense to him than a scripted approach. He has contributed significantly in implementing context-based testing in his work and has also written couple of white papers on the same. He is an active member of Testing Center of Excellence at Deloitte and contributes significantly there.

Venkat Moncompu is a manager with West Monroe Partners, LLC where he is advocating quality by design in the software development projects that he engages in. Venkat has 15 years of experience in the software industry and interests include quality management, agile test automation, user experience and design thinking. Venkat has a master's degree in Engineering from Arizona State University. Venkat has published articles and spoken at various forums on Quality Assurance in Information Technology. West Monroe Partners, LLC is an international, full-service business and technology consulting firm focused on guiding organizations through projects that fundamentally transform their business.

Copyright. Shaham Yusuf & Venkat Moncompu, 2012.

1. Introduction:

In the ancient world, some 300 years ago, there was a firm belief that all species of swans were white. Every bit of empirical evidence confirmed this to be true. But in the late 17th century, the discovery in Australia of a swan species that was completely black destroyed any previous notion that all swans are, *and must be*, white. It hardly mattered anymore that thousands of years of recorded history had only produced the undeniable truth that swans were white, because with the first sighting of a black swan, the undeniable truth was rendered false, says Nassim Nicholas Taleb, the famous author of the book "The Black Swan: The Impact of the Highly Improbable" (Taleb, 2007).

Taleb defines black swans by three attributes. First, as the outlier from our traditional expectations; second, as the one which carries high impact and lastly, as the one which is *possible* though not predictable. We consider Exploratory testing as the black swan of software testing.

Exploratory testing approach is seldom considered in our test planning as it is the outlier of our scripted testing approach which we take for granted. Exploratory testing carries high impact and can exponentially increase the defect detection rate. Further, with exploratory testing, we get a chance to look at issues during retrospection which does not happen in scripted testing early, thereby reducing defects from leaking to User Acceptance Testing (UAT). For an example, in scripted approach we usually tend to stick to the pre-defined test cases whereas exploratory testing helps to look back and learn from, on-the-fly while testing.

Taleb says one single sighting of black swan can invalidate the belief that all swans are white. This presentation is such demonstration of the black swan of testing which might help us to shift our mindset from the granted belief of scripted testing.

Exploratory testing may not always be applicable in certain situations e.g. compliance applications / testing medical devices etc. which have strict requirements and standards. However while testing those applications too, testers do have a '*tendency*' to look beyond the written steps or to look at certain 'other things' happening on the system. This 'tendency' or the 'other things' is a simple example of exploration. The key idea is to tap this 'tendency' in a way which benefits the application as well as stakeholders. In testing applications having strict requirements and standards, the exploration may be less compared to testing other types of applications. We are trying to showcase a blended approach, between perfect scripting and pure exploration, and calling it as Alternative Test Design Techniques.

1.1. Case Study (Traditional Approach):

Now look back and think about the testing approaches you have seen or you have followed in the history of your career. We tend more-or-less to follow the same old scripted or documented test case based approach and our belief is firm that it alone works, no matter what type of application we are testing. We bias this belief on the three pillars of 'Scripted Testing'. First is the execution accountability, second is traceability and third is detailed steps having expected results.

While working on a large and complex project and following the traditional scripted testing approach, we have captured the following metrics along with some key observations.

1.1.1. Test Design:

Test Cases Created	Test Case Complexity	Average time spent *Creation* of	Total time spent on Test Case	Average time spent *Review* of	Total time spent on Test Case	Total Time spent on Test

		each Test Case (hours)	*Creation* (hours)	each test case (hours)	*Review* (hours)	Design (Hours)
706	Simple	1	706	0.5	353	1059
1178	Medium	2	2356	0.75	883.5	3239.5
472	Complex	3	1416	1	472	1888
Total						
2356			4478		1708.5	**6186.5**

Some notes around above metrics:

Some hours were spent on requirement study, which is separate from the total test design time shown above.

Key Observation: Significant amount of project time, 6186 hours, was spent on Test Design. With the team of 12 resources (including test lead), we spent over 3 months of time in test design phase (considering 45 hours per week per resource).

1.1.2. System Integration Testing (SIT) Defects:

Test Cases Executed	Test Case Complexity	SIT Defects Detected by Test Cases	SIT Defects Detected on Ad-hoc basis (does not maps to a test case directly)	Total SIT Defects Detected
706	Simple	69	65	134
1178	Medium	104	96	200
472	Complex	24	39	63
Total				
2356		197	200	**397**

Key Observation: After going through robust test design phase, about 50% of the total SIT defects were detected on Ad-Hoc basis, i.e., they were tangential to the test cases.

1.1.3. SIT v/s UAT Defects:

Number of SIT Defects	397
Number of UAT Defects	50

Key Observation: There was significant defect leakage to UAT ((50/397)*100 = 12.5%).

1.1.4. Some overall observations:

We observed that after spending more than 3 months of time in Test Design phase, we discovered lots of the defects which were valid ones but not directly or explicitly related to the test cases per se. The test cases indeed helped the team to get closer to those defects though not directly. This made us take a step back and see if our Test Design was ineffective though those test cases were reviewed and approved by domain experts!

After doing extended analysis of the designed test cases as well as analysis of those defects, we found that most of the defects being detected were tangential to the pre-designed test cases because the pre-designed test cases are mere interpretation of requirements linguistically; whereas during testing we questioned the product from end user's expectation or perspective.

On top of the high defect leakage of 12.5% UAT, we realized we were too caught up in the perfection of test design. We were quite taken aback by the large number of so called 'non-test cases' defects which led to defect leakage to UAT. Maybe, we were so confident and comfortable with the detailed approach i.e., the test case based approach that we never felt the need to think in any other manner or of a better approach!

1.2. The trigger that led us to change our testing approach:

In the course of the project many events happened: we missed important bugs, bugs were found during UAT, some in production too and we got bugs at such a later stage that we challenged our so-called *robust* test design because we seemed to have missed bugs though we spent huge amount of time on creating detailed test cases and up-front planning.

Though these events are something that could not have been predicted in the traditional scripted approach, human nature compels us to create explanations for the occurrence of these events making it explainable and, in retrospect, predictable. In such situations we sometimes may say, *it is out of scope of our test case* etc.; but the client were not satisfied!

We make 10 years projections on oil prices, economy, social events, political affairs etc., without realizing that that we cannot even predict these for next summer! All our prediction failed terribly on major unpredictable events like financial crisis of 2009 or the 9/11 attack.

On a very similar note, we design amazing test cases with explicit steps prior to the creation of the application. While executing those test cases we are strongly biased by the steps and hence concentrate on the events which are mentioned in the steps or the expected results only. What about millions of other things happening in the application during the same time? Do we focus on them? Do we de-focus from them? These would result in defect leakage.

> "In practice, most testing that people actually do probably sits in the middle, somewhere between pure exploration and perfect scripting. My bias is that most of the best testing sits a lot closer to the exploratory side of that continuum" – ***Cem Kaner***

In a traditional approach (i.e., scripted testing), when a tester executes a pre-defined test case, he/she also looks into '*a day in the life*' type of tests. Putting the system through these conditions that is not scripted is nothing but *Exploratory Testing*. If these are practiced in a consciously structured manner then the results can be very impressive; as was in our case. The other pitfall of running scripted tests over-and-over is that these tests become ineffective over the repeated runs analogous to following in someone else' footsteps in a minefield which has a much lower probability of actually finding anything new.

2. What is Exploratory Testing?

Exploratory testing is an approach where tester investigates and finds various information about the product, which may matter to stakeholders. The tester is not driven by the pre-defined test cases, rather has control over the test designs during execution and uses the findings to design or vector subsequent tests.

Exploratory Testing is not procedurally structured, rather it is cognitively structured, says Michel Bolton (Bolton, 2008). It is like a chess game. We revise our plans after seeing the move of the opponent (application behavior). Yet all our plans can be changed after one unpredictable move of our opponent. Can we write cases for the chess move beforehand? Even if we write the exhaustive cases (millions of possible cases) then which case to apply when depends on the context and is a cognitive judgment.

Exploration is the basic nature of any human being. Human are curious by nature and are interested in investigation, examination and questioning. The dictionary defines exploration as investigation and examination of unknown regions.

2.1. Challenges in Implementing Exploratory Testing:

Primarily there are 4 typical challenges which make the traditionalists reluctant to adopt Exploratory Testing formally in engagements.

1. *Test Design:* Usually people think that test designing is poor as it is an exploratory/ad-hoc approach hence may affect the project deliverable.
2. *Accountability:* The word exploratory gives an impression to layman that accountability cannot be established hence they are reluctant to adopt it.
3. *Learning styles:* It is a usual notion that heavy documentation of pre-defined test cases will help the team members to learn the application quickly but exploratory approach can come in the way of knowledge transition.
4. *Negotiated Contract:* Negotiated contract with Client comes with many pages of requirement document for which we need to showcase the coverage.

2.1.1. Challenges addressed:

1. Test Design:

Defect leakage metric shows continued bleeding of defects even when test coverage is reportedly 100% of stated requirements. Short-sighted test managers rely excessively only on the stated requirements. This myopic view only adds to the poor quality as requirements are one-dimensional and static whereas user needs are multi-faceted that cannot be represented to meet all those needs. Exploratory Testing gives the tester this degree of freedom to transcend this gap to look at the application in a contextual perspective beyond the stated requirements. Exploratory Testing defines structure to the practice of testing that makes it risk-focused, value-driven and effective. The metrics we have gathered show a reduction in defect escape rate that shows the value-focus of Exploratory Testing.

Exploratory Testing can overcome the pitfalls of minefield analogy as articulated by James Bach. In feature driven development methodology for e.g., the risk profile keeps changing and test profiles also have to reflect these changes. Through exploratory testing, we design tests dynamically based on the current context continuously as the tester gains insight and feedback from the system. These result in effective testing of the system compared to those which are designed based on requirements linguistically.

> "Highly repeatable testing can actually minimize the chance of discovering all the important problems, for the same reason stepping in someone else's footprints minimizes the chance of being blown up by land mine." – **James Bach**

Our data demonstrates that the effectiveness of scripted tests decreases dramatically as a function of time (running same tests over and over again) across product releases. And since the adoption of Exploratory Testing in tandem with scripted testing, our data show significant improvements in the number and value of defects reported, which is also appreciated by the stakeholders.

2. Accountability:

In traditional or scripted testing the accountability is maintained by passing or failing every pre-defined test case. System behavior cannot be just evaluated objectively using pre-defined test cases alone.

In contrast, exploratory testing approach is more of a subjective assessment of the system where the tests are designed and executed based on sapience and learning to use the system (Bolton, 2009). One of the best ways to maintain accountability in exploratory testing is to practice session based testing (Bach, 2000). Disciplined exploratory test sessions are far more effective than a simple execution of pre-defined test case. Nowadays Exploratory sessions are also supported by many tools.

In the traditional approach, in a way testers are forced to follow a premise that '*this is what we are going to do, irrespective of the way the application behaves at this time*'. In contrast the exploratory approach implies '*This is what we have done*'. Now, we leave it on you to decide which is more accountable and effective?

3. Learning styles:

Scripted approach relies less on the cognitive skills of the testers in terms of continuous discovery and learning whereas Exploratory Testing relies on heuristics and effective learning techniques. The more we rely on the steps of test cases to learn about the system, the more easily we forget about the system. In contrast if we learn the system by process of exploration and discoveries, less is the chance to forget about it – *because we discovered about it, steps didn't tell us!*

4. Negotiated Contract:

It is typical notion in industry that coverage can be shown by number of requirements pass/failed. Well, what about the requirements which are still evolving? What about missed requirements? In Exploratory Testing, we can still show coverage of the documented requirements by means our test execution. This is most favored by agile development teams who adopt just-enough and just-in-time requirements.

3. The Alternative Test Design technique:

We like stories, detailed steps and documentation and we like to simplify them, i.e*., to reduce the dimension of matters*! It is because all of these look very good. There is a flip side of these though. One of the ways in which human beings are blind to the occurrence of unpredictable has to do with something called the "narrative fallacy," a problem that illustrates our need to turn abstract concepts into understandable stories. We write detailed test cases from the abstract application i.e., the requirement documents and we expect to find all the vital defects of the application from those test cases though the context during execution may be different. In short, how it is possible to understand the properties of the unknown (the application), based on what is known theoretically (the test cases and the requirements)?

After extensive analysis, thought and brainstorming, we realized that test design has to be light weight so as to complement user expectation as well as evolving requirement specifications.

We toyed with several ideas, like,

- Writing high level test cases
- Not writing test cases at all and just relying on session based exploratory testing
- Using requirement documents only as our guide while testing
- Leveraging use cases and user stories to derive ad-hoc test cases while testing
- Stop toying and get back to the traditional approach

Soon we realized that there could not be a single alternative; we have to come up with an approach which leverages best of all approaches and should be in-line with our project – which means it should suit the context of our project.

We need something in the form of guidance heuristics for our test cases. We knew that some high level test design is required for our needs. But the nature of our project demanded mix of things. We boiled them down to the following set of light weight test design techniques,

1. Scenarios – *scenario based testing complemented by test execution narration*
   - We used high level test scenarios for designing complex business rule type of tests

2. Flow Diagrams
   - We used flow charts for designing end-to-end type of tests

3. Mind Maps
   - Mind Maps were used to jot down spontaneous ideas which flow during JAD (Joint Analysis & Design) sessions and requirement analysis phase.

4. Cheat Sheets
   - We used cheat sheets for existing/old modules, primarily to regress the system.

5. Checklists
   - Checklists were extensively used to design nitty-gritty types of test e.g., Date should be in GMT, Text should be Ariel Black, size 14 etc.

We have not coined these test techniques but have merely leveraged them in our project. There can be other test design techniques as well, but the above five worked well in the context and nature of our application. The key idea is that test design should be at a High-Level rather than detailed step-by-step so that testers get a freedom to explore and at the same time they are more responsible for testing as opposed to just following the steps.

All of the 5 test design techniques which we have used in our project encourages testers to document tests at a high level. They have different purpose in the context to type of test, for example; 'Flow Diagrams' were used for end-to-end type of tests whereas checklists were leveraged to ensure that we are missing nitty-gritty type of requirements.

We highly encourage you to choose your high level test design techniques after careful analysis in context to the nature of your project.

3.1. Deep Dive of Test Design Techniques:

1. **Scenarios:** A scenario is a hypothetical story, used to help a person think through a complex problem or system. "Scenarios" gained popularity in military planning in the United States in the 1950's. Scenario-based planning gained wide commercial popularity after a spectacular success at Royal Dutch/Shell in the early 1970's.

   A kid explores the world by tasting everything (Putting things in mouth) and learns which is what. But we sometimes instruct them what to do and what not to. We provide some direction to them so that they can be guided toward a better understanding.

   It seems that Scenario based testing exists in the normal testing life cycle processes and maps Use Case to Scenario and then to Test Cases. In our approach we stop at Use case to Scenario and move towards execution.

   While we do agree to this it does not adhere to the processes that are required and defined for test case based testing which are very much essential while delivering large complex projects. Here we have taken steps to make sure people endorsing quality believe in our "black swan" of testing. In this

approach we have designed processes and documentation which will support answers to all stakeholders who define and endorse quality.

Here are some characteristics of an ideal scenario:

1) The test is based on a story about how the application is used, including information about the motivations of the people involved.
2) The story is motivating. A stakeholder with influence would push to fix a problem that fails this test.
3) The story is credible. It not only could happen in the real world; stakeholders would believe that something like it probably will happen.
4) The story involves complex use of the application or a complex environment or a complex set of data.
5) The test results are easy to evaluate. This is valuable for all tests, but is especially important for scenarios because they are complex.

In our case we have customized it to suit our project needs and context. We highly encourage you to customize it to suit your projects. No matter which format you follow the basic idea remains the same – to have a short story or objective of the scenario followed by a one line process and an open ended output.

In our case we refer to the objective as the story of the scenario. We make the objective motivating and credible based on the knowledge of the application and the business. We document complex situations and state transitions by matrix of combinations and other accelerators which suits the context.

Below is a typical test case having detailed steps and is followed by a typical scenario for the same test condition. The below comparison shows how a high level scenario handles the test condition in few lines whereas the test case has more than 8 detailed steps for the same situation.

Description and Test Case Name	Step Name	Description (Design Steps)	Expected Results
Objective:To check whether the "Show Changes Only" checkbox is functional when user navigates to ABC XYZ Tab of an Add XYZ Change request **Pre-requisite:** **1.** User should have ABC rights. 2. System has ABC state PRODUCT ABC_2.2_Defect123456_ABCManagement_ViewChangeRequestPagesShowChangedRowsdata_AddABCD.CR_XYZ Tab_PT_TC_0052	**1. Login**	1. Login to the application	1. User should be able to login.
Objective:To check whether the "Show Changes Only" checkbox is functional when user navigates to ABC XYZ Tab of an Add XYZ Change request **Pre-requisite:** **1.** User should have ABC rights. 2. System has ABC state	**2. Search & Navigate**	1. Enter at least 2 characters in the "Search for" field in the basic search page. 2. Click on "GO" button	2. The ABC summary page should be displayed.
Objective:To check whether the "Show Changes Only" checkbox is functional when user navigates to ABC XYZ Tab of an Add XYZ Change request **Pre-requisite:** **1.** User should have ABC rights. 2. System has ABC state	**3. Add**	1. Click on " Add ABC" button. 2. Fill in all the information of the following: 1)ABC details 2)Address 3)XYZ and Contacts.(Note: Ensure ABC is added for the XYZ) 4)Classification. 5)Comments. 3. Click on Save, Save & Close.	1. The Add ABC Page will be displayed with the following tabs: 1)ABC details 2)Address 3)XYZ and Contacts.(ABC should be added for the ABC) 4)Classification. 5)Comments. 2.All the details should be added and saved successfully resulting in an ADD ABC CR for that ABC. Clicking on " Save & close" should return to the Basic search Page.
Objective:To check whether the "Show Changes Only" checkbox is functional when user navigates to ABC XYZ Tab of an Add XYZ Change request **Pre-requisite:** **1.** User should have ABC rights. 2. System has ABC state	**4. Items in review Page.**	1. Navigate to Items In review Page , search for the batch and view it. 2. Click on Add sample button. 3. Fill in all the mandatory fields and save the CR	1. User should be able to search for the batch. Clicking on the view link should take the user to the CR details Page that consist of the add ABC CR. 2. An add sample page will displayed. 3. All the sample related details should be saved resulting in an add sample CR.
Objective:To check whether the "Show Changes Only" checkbox is functional when user navigates to ABC XYZ Tab of an Add XYZ Change request **Pre-requisite:** **1.** User should have ABC rights. 2. System has ABC state	**5. View**	1. Now click on the VIEW link corresponding to the Add ABC CR type	1.View Change Request Page will be displayed
Objective:To check whether the "Show Changes Only" checkbox is functional when user navigates to ABC XYZ Tab of an Add XYZ Change request **Pre-requisite:** **1.** User should have ABC rights. 2. System has ABC state	**6. Show Changes check box**	1.Now check to see the SHOW Changes Only checkbox 2.Naviage to XYZ tab. 3.Navigate to ABC details, Address, classification and codes and review tab. 4.Click on " Cancel" button.	1. Show Changes Only checkbox must be displayed. The checkbox must be unchecked by default 2. The show changes check box should not be visible in Responsible and Party tabs. 3. The show changes check box should be visible in ABC details, Address, classification and review tabs. 4. Clicking on Cancel button should return the user to CR details page.
Objective:To check whether the "Show Changes Only" checkbox is functional when user navigates to ABC XYZ Tab of an Add XYZ Change request **Pre-requisite:** **1.** User should have ABC rights. 2. System has ABC state	**7. Move to ABC state**	1. Click on "Move to Global reviewer 1" button 2.click on the view link of the respective batch from the Items in review Page. 3. Repeat Step 5 to Step 6.	1.User should be able to move the batch to ABC level. 2. The CR details page of the ABC state batch will be displayed 3. Verify Expected result of Step 5 and Step 6 respectively.

This is a typical test case having detailed steps and expected results. As discussed earlier it may limit a tester's ability to detect bugs.

Scenario 01	
Objective:	To verify the "Show Changes Only" checkbox is functional when user navigates to ABC XYZ Tab of an Add XYZ Change request
Requirement Details	
Related Defects	123456
Input	The ABC summary page with add sample page is displayed
Pre condition:	1. User should have ABC rights. 2. System has ABC state
Process	Perform actios on "Show Changes Only" checkbox
Output	The checkbox functions and shows the changes
Secondary Expected Results	Other functions are intact
Additional information (for testing)	There was a related defect in previous release, ID 123456
Checklist	1. Verify in view mode also 2. Verify in all states

Matrix				
Initial State	Data Input	Condition	Process	End State
ABC State	Add Sample Page	With abc data	Add	Gets added
XYZ State	Add Sample Page	With xyz data	Remove	Gets removed

A typical Scenario

3.1.1. Execution Narration for Scenario Based Testing:

Execution Narration is a conceptual method to record the contextual test cases designed and executed during the scenario based test execution. In other words scenario based test execution is made accountable by proper Execution Narration.

3.1.2. What to narrate?

As our scenarios are high level and have a one line process step, it becomes important for us to know what has been executed. We need to understand the need of our stakeholders and narrate the required tests performed.

3.1.3. A typical 'Execution Narration' of a Scenario based test execution

Scenario 1 To test ABC report generation

Session started at 1 PM.
Saw the UI of all the search criteria (GUI only).
Verified start time (GMT). - Data Source is 0 (*pre existing since release 2.1)*

'Include Inactive' is not checked by default - *its fine as per specs*
Session Interrupted by a Phone call

Downloaded the file
File of 1.9 MB took approx 3 minutes to generate – longer than previous iteration
Generated file on the export page is correct

- Generated big files. 225 MB and 120 MB. in both cases file does not open with full data in excel. -- seems fine, limitation of excel I guess – Need to report as information

Test took 20 minutes
Session ended at 1.30 PM for lunch

Session resume at 2.50 PM

- Initial Approval Date has 00 as GMT time for all values prior to 2005. - *may not be an issue, need to discuss*

Last modified date is not pulled into export file - Its a Defect - Can't confirm if its pre existing issue or not, since history refresh is not working.

- Parsing error is present in export file. – need to report as low

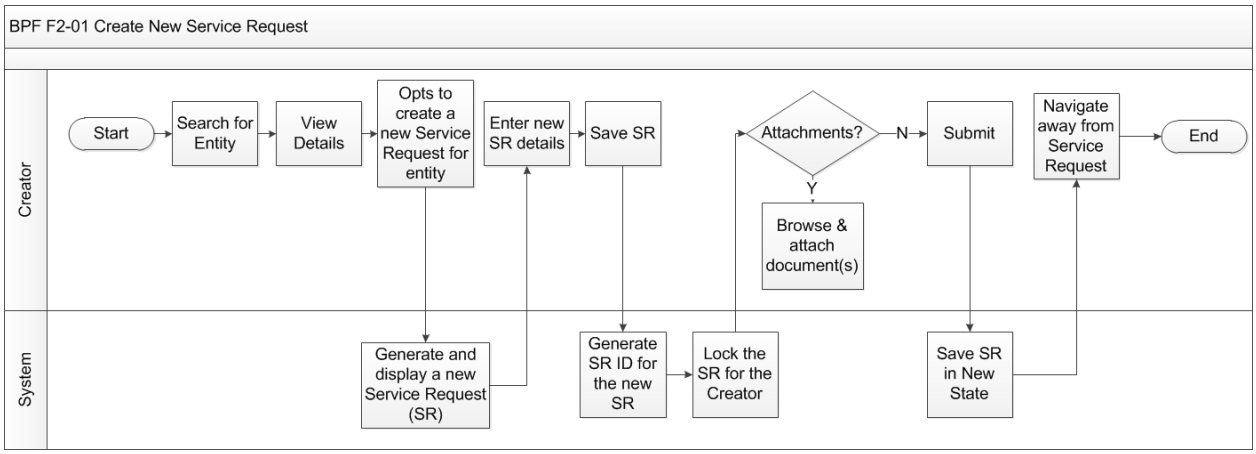
- Exported with single criteria & 2 attributes
- Then exported by giving 2 criteria with single attribute in each & then export with 1 criteria & 2 attributes. Entity is pulled, as Found that it works on AND logic – *works fine*

- SIC code is also appearing. Updated the SIC code to a new value. – update value is pulled.

- Multiple identifiers are appearing separated by pipe. – *need to confirm is pipe or comma should be the separators*

Test took 2 hours
Session ended at 4.55 PM

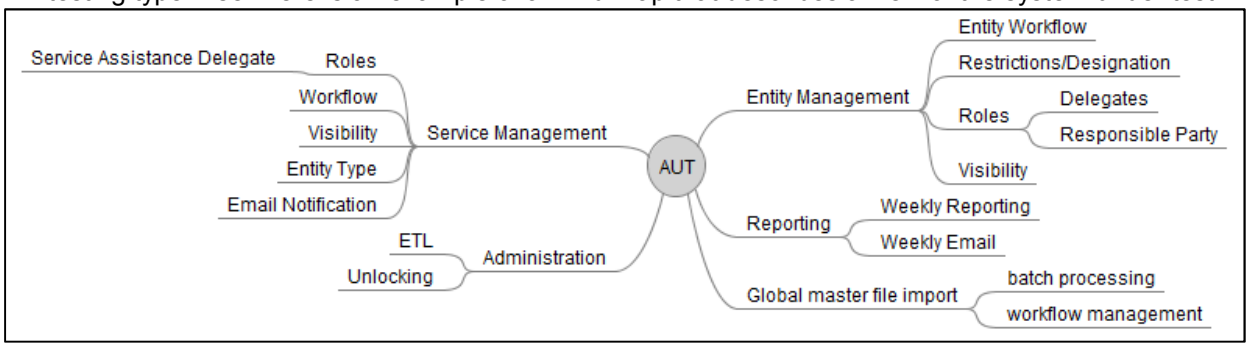
2. **Flow Diagrams:** We created flow diagrams using MS Visio to design the end-to-end types of tests. These were the test where we are NOT concerned about a module/functionality working in silo but multiple functionalities/modules should work in conjunction.



3. **Mind Maps:** Mind maps are great aid to spontaneous test idea generation. The open ended nature of Mind Map allows to club ideas in logical manner. The best times to create Mind Maps are during JAD (Joint Analysis & Design) sessions or during team brainstorming sessions. We have gathered some great ideas using Mind Maps and leveraged them during testing.

   They are unique to each tester as it is for any user of mind maps. It helps to organize the thought process that is proven to be non-linear and therefore not friendly to the way languages are written. Mind maps provide the spatial dimension to facilitate the thought process. Ideas by association and spatial reference are the keys to assimilating thoughts which are otherwise not comprehensible. Mind maps provide just the right framework for exploration.

   Depending on the nature of your project, you may create Mind Maps functionality/module wise or testing type wise. Here is an example of a mind map that describes a view of the system under test.



4. **Cheat Sheets:** Cheat sheets are useful to test a known functionality or module. We basically used them to regression test the existing modules. We created separate cheat sheets for each module, mentioning the following,

   A. Brief summary of that module
   B. Types of suggested tests for that module e.g., Concurrency Test, Roles & rights Tests etc.
   C. Must-execute tests e.g., any module specific calculations or validations etc.
   D. Some history of the module
      i. Know 'Open' defects (if any)
      ii. Types of defects found earlier in this module

5. **Checklists:** Different types of checklist can be used depending on the nature of the project. We use a checklist which covers the explicit requirements, and we need to 'check' that they work in every iteration / release. We wanted to check this in every iteration as they were working before and we want to confirm it newer iteration(s). Here is a typical Checklist,

Check List		Description	Iteration1	Iteration2	Iteration3	Iteration4	Iteration5	Iteration6
	#							
Text	1	Headings 24, Arial Blue	✓	✓				
	2	Sub headings 16, Arial, Black	✓	✓				
	3	Text 14, Arial black	✓	✓				
	4	All text are localizable	✓	✓				
	5	Roman, Hebrew, Chinese are accepted	✓	✓				
Date	6	Date in GMT	✓	✓				
	7	Start date is never greater than end date	x	✓				
	8	End date is never more than current date	✓	✓				
General	9	Bread crumb is always clickable, expect the current page (last one)		✓				
	10	Bread crumb is Home > module > sub module > page	✓	✓				
	11	Enter key begins search	x	✓				
	12	Sequence of Tab key is first field to last field	✓	✓				
	13	Escape works as cancel	✓	✓				
	14	Help menu is present on each page	x	✓				
	15	Buttons have key commands	✓	✓				
	16	All buttons are all of similar size and shape, and same font & font size	✓	✓				
	17	Mouse and keyboard action are consistent in all pages	✓	✓				
	18	Session extension appears after 20 minutes	✓	✓				
	10	Session extension appears twice and then system logs out	✓	✓				
	20	Checking the check box refreshes the screen	✓	✓				
	21	Lists are scrollable	✓	✓				
	22	Arrow key works in list	✓	✓				
	23	All windows have a consistent look and feel.	✓	✓				
	24	Numeric field does not accepts other characters	✓	✓				
	25	All numeric fields have field length of 20	x	✓				

Many people say that checklist and scripts are more or less same. But it is not. A checklist is an open ended item, whereas a script is a close ended item. You may validate an item of a checklist in any number of ways. But in scripts, you would follow the specified steps. The details can be referred in "value of Checklist by Cem Kaner" (Kaner, 2008) in which Cem Kaner describes how checklist differs from scripts.

4. Summary of test Design Techniques:

We are not saying that the above five test design techniques are the only alternative for the traditional test design. We are trying to showcase how these 5 techniques helped us to design robust tests in our project. We encourage you to try similar light weight test design techniques which would suit the context of your projects.

4.1. Case Study (Alternative Test Design Approach):

Here are some metrics which we gathered post implementation of Exploratory testing with light weight test design, in our project.

4.1.1. Test Design

Test Design - Artifacts	Number of Test Artifacts	Average time spent *Creation* of each Test Artifact (hours)	Total time spent on Test Artifact *Creation* (hours)	Average time spent *Review* of each test case (hours)	Total time spent on Test Case *Review* (hours)	Total Time spent on Test Design (Hours)
Scenarios	481	1.5	721.5	1	481	1202.5
Flow Diagrams	55	1.5	82.5	0.5	27.5	110
Mind Maps	20	2	40	1	20	60
Cheat Sheets	15	1	15	0.5	7.5	22.5
Checklists	27	1	27	0.5	13.5	40.5
Total	**598**		886		549.5	**1435.5**

Key Observation: Considerable amount of Test Design time is saved due to light weight test designing.

4.1.2. SIT Defects

Test Artifacts Executed	Test Artifacts Executed	Defects Detected by Test Artifacts	Defects Detected on Ad-hoc basis (does not maps to any of the test artifact directly)	Total Defects Detected
Scenarios	481	289	11	300
Flow Diagrams	55	23	1	24
Mind Maps	20	34	6	40
Cheat Sheets	15	3	8	11
Checklists	27	52	0	52
Total	**598**	401	26	**427**

Key Observation: Most of the detected defects were the outcome of the execution of the high level test artifacts. Only 6% of the total defects were detected on Ah-Hoc basis; thus the designed artifacts were leveraged completely and effectively.

4.1.3. SIT v/s UAT Defects

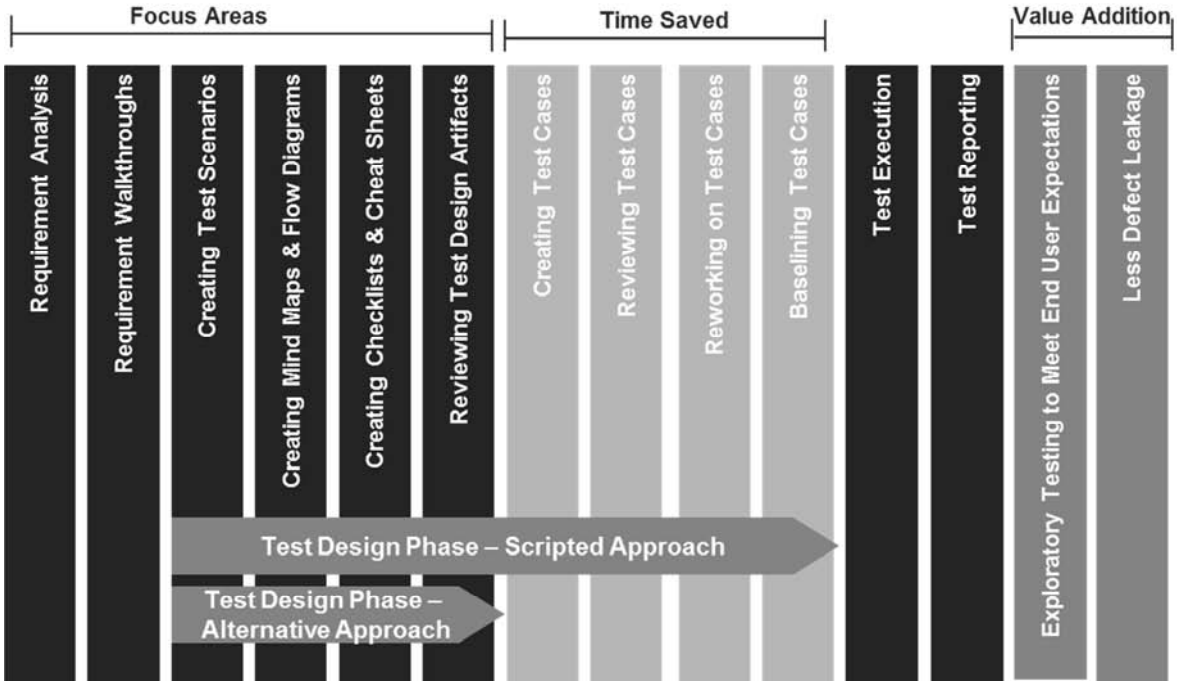
Number of SIT Defects	427
Number of UAT Defects	10

Key Observation: Defect leakage was on 2.3% as the high level test artifacts allowed testers to explore like end users.

Note: Client used pre-defined test scripts for UAT prior to implementation of our alternative Test Design Techniques as well as after the implementation.

4.2. Some Overall Observations:

Lightweight test design saves considerable test design time as well as reduces defect leakage.



4.2.1. Some Challenges during implementation of Exploratory testing using Alternative test design techniques

Some of the challenges during implementation were related to management, team members and efficiency of execution. We have described below those challenges and their possible solutions.

Leadership & Management	▪ You need to convince your project stakeholders by showcasing the benefits of this approach ▪ A pilot should be conducted prior to implementation in large scale projects ▪ Design Test Strategy and Test Plan to suite the context of Scenario Based approach
Test Team	▪ Change in mindset of the resource is required as they would be transmitting to high level execution from an approach which was having detailed steps ▪ Team members need to sharpen exploratory testing skills by regular practice
Effectiveness & Efficiency	▪ During initial phase effectiveness may be bit less but it would increase drastically once the process is matured ▪ The lessons learnt and good practices can be leveraged to other projects of the organization for efficient execution of the later projects

4.3. Do it yourself:

We thank you for spending your time in reading the benefits of exploratory testing. We highly encourage you to pilot it in your organization / project and see the benefits for yourself. A few things to keep in mind which will help your organization become more successful.

Ensure that the stakeholders are well informed and know the pros and cons of the effort:

1) We suggest you conduct some competency development sessions for your team members to enhance their exploratory testing skills. This can be done by organizing few time bound testing sessions and giving them some open source applications to test and discuss their findings.
2) Proper context based test plan should be prepared and should be updated regularly based on the dynamics of the project
3) While creating / designing scenarios the checklist concept should be used extensively so that you do not miss any requirements of the negotiated contract
4) Ensure to maintain proper execution narration for the test execution accountability and share the narration to the stakeholders on daily basis along with the daily status report
5) Conduct some short sessions within your team to explain effective narration writing technique, prior to the execution.
6) As you understand the benefits of Exploratory Testing, please be aware of some of the cons too:
    a) Exploratory tests call for testers to be willing and open to learning from the system as opposed to requirements-based expectation from the system.
    b) There is an expectation that testers are familiar with the business function and what the user wants to achieve with the task being implemented using the system.
    c) Exploratory testing calls for clear oversight and definition of objectives from the test. The retrospective from the test sessions are very important to ensure these tests are effective.
    d) The tester is expected to understand the role of each user of the system and the tasks that are needed to be performed by the system under test.

5. Conclusion:

Our findings of improved defect detection by as much as 12% on our project compares well with the gains other studies reported (Itkonen, Mantyla & Lassenius, 2007). Some other projects adopting exploratory testing that have reported savings of up to 30% in the testing budget over their scripted counterparts. In lean practice of software development one of the ways of identifying waste is to use a value mapping of all the process activities. If the same is done for testing processes, the scripted testing shows little value being added from the up-front activities of documentation and review whereas exploratory testing shortens the time to adding value to the quality. These findings provide compelling evidence that testing processes can only improve with the adoption of exploratory testing. We urge the skeptics out there to give exploratory testing a second chance!

References:

1) *The Black Swan* by Nassim Nicholas Taleb, Random House, 2007
2) *An Introduction to Scenario Testing* by Cem Kaner
3) *Scenarios: The Art of Strategic Conversation* by Kees van der Heijden, Royal Dutch/Shell's former head of scenario planning, Wiley, 1996
4) *Value Of Checklists* by Cem Kaner, CAST 2008 presentation
5) *A tutorial in Exploratory Testing* by Cem Kaner
6) *The Nature of Exploratory Testing* by Cem Kaner

7) *Rapid software Testing* by James Bach
8) *Session Based Test Management* by James Bach, STAR West 2000
9) *Exploratory Testing: Design, Execute, and Learn* by Michael Bolton, QUEST 2008
10) *Testing vs. Checking* by Michael Bolton, Agile 2009
11) www.testertested.blogspot.com by Pradeep Soundararajan
12) Juha Itkonen, Mika V. Mantyla and Casper Lassenius, *Defect Detection Efficiency: Test Case Based vs. Exploratory Testing*, First Intl. Symposium on Empirical software Eng. and Meas. IEEE Comp. Soc., 2007
13) Lisa Crispin and Janet Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*, Addison-Wesley Professional, 2009
14) James Bach, *To repeat tests or not to repeat*, http://www.satisfice.com/blog/archives/24
15) Cem Kaner, *Exploratory Testing*, Keynote address at QAI conference, Nov, 2006

Avoiding Overkill in Manual Regression Testing

Lisa Shepard
Lshepard@webmd.net

Abstract

Software QA Analysts take great pride in thoroughly testing every aspect of a product's functionality. As complexity and variability of a product increases, so does the need for more thorough testing to ensure excellent quality and few bugs. At WebMD, we have found that one of our biggest impediments to thorough testing is over-documentation of all the tests that have already been run. To combat this, we encourage our QA Professionals to think critically about what manual tests to document. More specifically, we ask them to think strategically about what manual tests NOT to document.

For most QA Analysts, it's a natural instinct to document everything that was tested: for posterity, for future regression testing, to prove their worth to their manager, or a variety of other well-intentioned reasons. This over-documentation not only wastes the individual test writer's time, but it creates a stockpile of indigestible artifacts that ends up having a negative effect on overall quality.

This paper gives concrete examples of when we should and shouldn't document a manual regression test. This information will help QA professionals, developers and managers who erroneously equate the quantity of tests with the quality of testing and inadvertently sabotage their organization's testing efforts in the process. This provides details about what *should* be included in a regression test. Just as importantly, it highlights those details that *should not* be documented.

Using what is learned in the paper, QA professionals will be better equipped to write manual regression tests that are robust, readable, and low-maintenance. In doing so, these QA professionals and their development teams will be in a position to see huge benefits of spending more time testing and less time documenting excessive details.

Biography

Lisa Shepard is a Lead QA Analyst at WebMD Health Services in Portland, OR, where she has been diligently squashing bugs and improving quality since 2003. Although Lisa began her official software career in the late 1990s through college internships and a job at a startup company, she has been using her "QA brain" since early childhood. Throughout Lisa's nine years at WebMD, her test writing style has changed dramatically, which can be seen by looking at some of her historical (and embarrassing) test plans from the past. Among other projects, Lisa is on a mission to help QA Analysts learn how to avoid some of the same pitfalls and mistakes that she has made that led to bloated test plans!

Lisa has a B.S. in Computer Information Systems from the University of Idaho, as well as a B.A. in Spanish. When she is not sitting in her chair at work, you will likely find Lisa wakeboarding, coaching others on health & wellness, or enjoying a peaceful evening on the back deck with her family and friends. Lisa lives in West Linn, Oregon with her husband and 4-year-old daughter (who, incidentally, is also showing signs of a QA brain!).

Copyright Lisa Shepard 2012

1 Introduction

When developing software, it is essential to run a multitude of tests to ensure the code is functioning properly and to find and fix as many defects as possible. There are often misguided intentions, however, about what to do with all of those functional tests *after* initial development is over. In some extremes, all the test cases are thrown away and no one ever tests that functionality again. In other situations, all of the test cases get saved under the guise of "regression tests." In this situation, the massive set of manual regression tests soon becomes too big to actually run on a regular basis, and just as if there were no tests recorded, this functionality never gets tested again.

As you can see, it can be just as harmful to over-document tests as it is to under-document them. In this paper, QA professionals will learn when it is appropriate to write a manual regression test and when it is better to leave the test undocumented. We will also learn what to include in a test case (and what to omit) to ensure that the test cases are effective but not overly fragile.

Please note that the target audience for this methodology is organizations that don't deal with matters of life and death. Although every company would desire zero bugs in the production code, there are certain organizations that devote extra resources and budgets to their QA effort because the cost of a bug (in lost lives or money) is too expensive to allow a single bug into production (airplane instrumentation, some medical equipment companies, etc.). Although software bugs in production are costly for every company, most corporations put a limit on the time and resources allotted for functionality and regression testing, and recognize that limiting the resources opens up the possibility of bugs slipping through the cracks. It goes without saying that these corporations want high quality code in their production environment, but the bugs don't result in loss of life or limb.

This paper will provide concrete examples of how to effectively document manual regression tests. After reading it and practicing some of the ideas set forth, QA professionals will be better equipped to write manual regression tests that are robust, readable, and low-maintenance.

2 Is there such a thing as too many tests?

In the beginning of my career at WebMD, I was responsible for in-depth testing of some new functionality. With every test that I ran, I documented the test and the resulting behavior. I found great security in documenting every piece of information I was learning. I used the tests to guide my current testing, to document the expected behavior, and to keep track of what I would need to test when doing regression testing with each release. After a couple of months and thoroughly testing a few new areas of functionality, I was already up to about 200 discrete tests. Within a year, I had a suite of about 1500 tests within about 20 sub-categories of functionality. Not only did I feel like I had done great work for WebMD, but managers encouraged their QA resources to do the same thing… to prove that we had fantastic test coverage.

2.1 What's the harm in writing a lot of tests?

As you can imagine, I didn't regression test every single one of those tests with every release cycle. There's no way I would have had time for that. But if there was ever a question about expected functionality, I could refer to them. After a little bit, a few of them became inaccurate due to changing functionality, but I could spot when that happened, right?

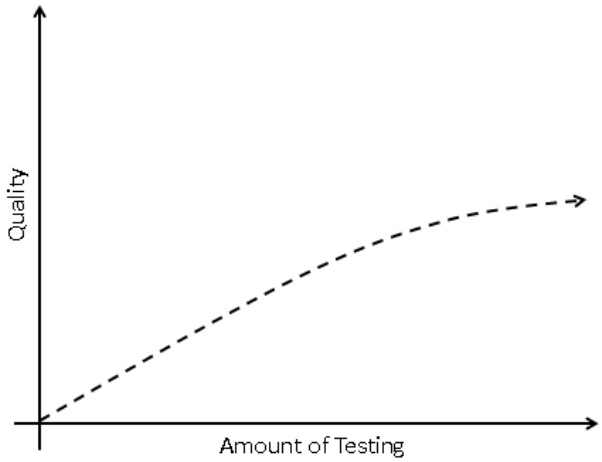
Fast forward about two years. I had been transferred to another project and someone else had been forced to muddle through my awesome collection of tests. Of course, they had no idea which ones could be trusted for accuracy. They didn't know the search terms or names that I used to find tests when I needed to refer to them. In fact, even though there was a lot of useful information in the test suite, the

entire collection had been moved to a “deprecated” folder. And the new QA team in charge of this functionality started all over again, creating its own suite of tests. The cycle had started anew!

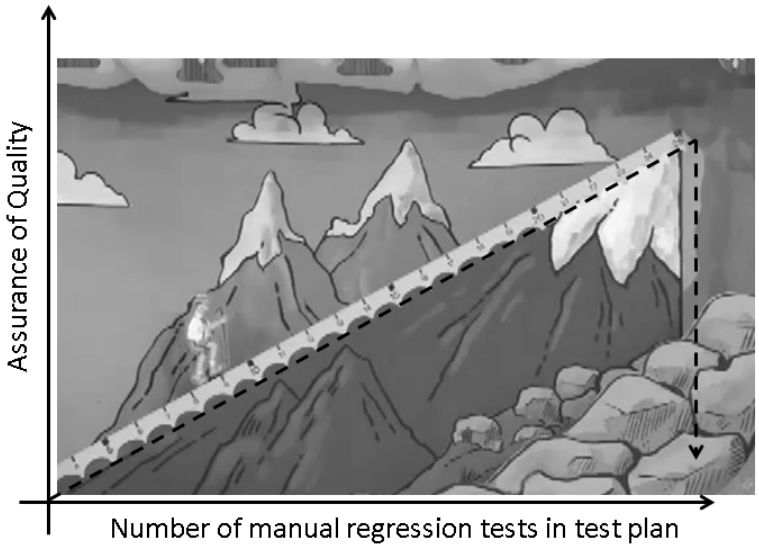
So in the end, not only did I inadvertently waste hundreds of company hours writing the test cases, but at the point that I crossed over the line into “too many tests to digest”, I rendered the entire suite of tests useless. I have watched countless QA teams do this on numerous projects, doing more damage than if they hadn’t written anything in the first place.

2.2 The relationship between tests and quality

We all know that testing is essential to ensure a high quality product. In general, the more you test, the better the quality will be, although this *does* taper off at a certain point and provide less of a return. Depending on the product being developed, each organization may have a different definition of what is *enough* testing, but the correlation between testing and quality can be described with the following diagram.



Notice that the horizontal axis is labeled “Amount of Testing,” not “Number of Tests”. In fact, when we start looking at the number of documented manual regression tests, I think of the graph as a mountain cliff. As we add manual tests to the regression testing plan, we increase the assurance of quality that we get with each round of regression testing. But as soon we’ve reached a critical point in the number of documented tests, the test plan is no longer manageable, maintainable or digestible. The entire test plan starts to get ignored and the assurance of quality doesn’t just decrease… it drops off a cliff.



When it comes to documenting manual test cases, our goal is to find a point where we have adequately documented the cases we plan to use for manual regression testing, but to document no more than that.

3 Why do we document tests in our regression plans?

There are many reasons why QA professionals document tests. Although there are one or two valid ones, most of the reasons for documenting regression tests are detrimental to overall quality. The QA professionals have good intentions, but those good intentions wind up resulting in bloated, unmanageable test plans. Here are some of those reasons and how they apply to the documentation of manual regression tests.

3.1 To document acceptance criteria

During the development and testing of new functionality, we create test plans to ensure we are delivering all of the features and functionality we are supposed to deliver. Whether by looking at product specifications or collaborating with a business owner on stories, we track tests for the golden path, edge cases, small units of functionality, and various combinations of settings.

As important as all of these tests are, it doesn't necessarily mean they all are worthy of being elevated to the status of "regression test." In a perfect world, where time is limitless, every single test that was run during initial development would be rerun release after release. But that world doesn't exist and time is limited, so we have to be smart about what gets included in the manual regression test suite.

Even though a test case was recorded during initial verification of product functionality, many of those tests should NOT be kept around for the long term. For this reason, it's important to be smart about how much time is spent on these tests that will eventually get thrown away.

3.2 To document functionality

Quite frequently, functionality isn't thoroughly recorded in a requirements specification or product document. As we create new functionality or fix bugs, QA's perfectionist tendencies want to ensure we have every single detail tracked. The last thing we want is for something to slip through that same crack in the future. Documenting the test gives us a false sense of security that we have patched up the crack.

Remember, if we write so many tests that it causes the entire suite to get thrown into the garbage after we leave the project, then we haven't done anything to increase the overall quality of the product.

3.3 To prove our value

We are rarely willing to admit that we are documenting tests simply to prove what a good job we did, however, there are organizations that erroneously equate a high number of test cases with a high quality of testing. If you are working in an environment where you think you need to write a lot of tests to show that you are actually doing work, stop what you're doing and address this with your manager.

3.4 To aide in creation of automated tests

Sometimes, it is necessary to document tests that you plan on automating in the near future. I would contend these tests need to have very little detail in them and they shouldn't be around for long. If these tests are frequently getting automated, then there won't be a glut of unnecessary tests clogging up the test plan. If the "Tests to be Automated" repository is always growing, however, then the mountain of artifacts is not worth the technical debt of creating and sorting through them.

3.5 To guide in regression testing

To many people, it might seem ridiculous to state that one reason to write manual regression tests is to help with manual regression testing. The reality is, however, that this is one of the few valid reasons to document a regression test. If you have no intention of running a regression test with every release, there is no reason to keep it around.

4 So how do we document just the right amount?

How do we achieve the right balance of documented manual test cases? Is there some magic formula that tells us what to document and what not to document?

The simplest way to find that balance is by following one rule:

All existing regression tests MUST BE TESTED during every major round of regression testing

When teams are forced to follow this rule, a few things happen:

4.1 We think twice about spending excessive time on writing functional tests

If there's a good chance that most of my individual functionality tests are going to be deleted or combined into a more conclusive regression test at the end of a sprint or the end of a release, then I am going to be much more strategic about how much time I spend on documenting those tests. I can document what is necessary to make sure I test all the cases I need to, but I don't need to ensure that it is in "archive" worthy format. I can spend more time testing and less time writing about what I just tested.

4.2 We can trust the information in the manual regression tests

Have you ever run across a test in a test repository and found that it didn't match up with current functionality? What normally happens in those cases? Rarely, the discrepancy highlights a bug that has just been introduced and the regression test has done its job of flushing out unexpected regressions in behavior due to new development.

The majority of the time, however, we find that the behavior had changed one or more releases ago, but the test just wasn't updated because nobody knew it existed. After the confusion is settled, no bug is entered and the test still remains unmodified. The cycle then repeats itself a couple of releases later when someone else finds the same "bug."

Once we are in a pattern where every test case is run with every release, we start to find fewer and fewer false bugs. When we do find a scenario where behavior behaves differently than the test case, it's likely that the unexpected behavior was introduced with the most recent development. At this point, we can decide if the test case needs to change to reflect new behavior or if the code needs to be changed to fix an unintended bug.

4.3 We continue to review our test plans with every release

As years go by, the number of test cases added to our test plans will continue to grow. Even when the test cases are written concisely and strategically, this may still result in the plan growing to a size that can't be manually regression tested with every release. At any point we find our departments unable to make it through all of the regression tests, we have two options: Delete some tests or allocate more time to regression testing.

In many cases, the appropriate response is to groom and delete the existing test cases. When this is the case, it is important to recall the visual of the hiker climbing up the cliff. By deleting those "excess" test cases, we are NOT decreasing quality. We are keeping that hiker from falling off the Cliff of Quality Assurance. We are actually helping the overall QA effort by deleting tests that no longer contribute to the quality of the product.

If someone makes a conscious decision to not run a set of tests because time is short, they are breaking the "Run Every Test" rule and need to remedy it by either deleting the test or by testing more.

5 What should be included in a manual regression test?

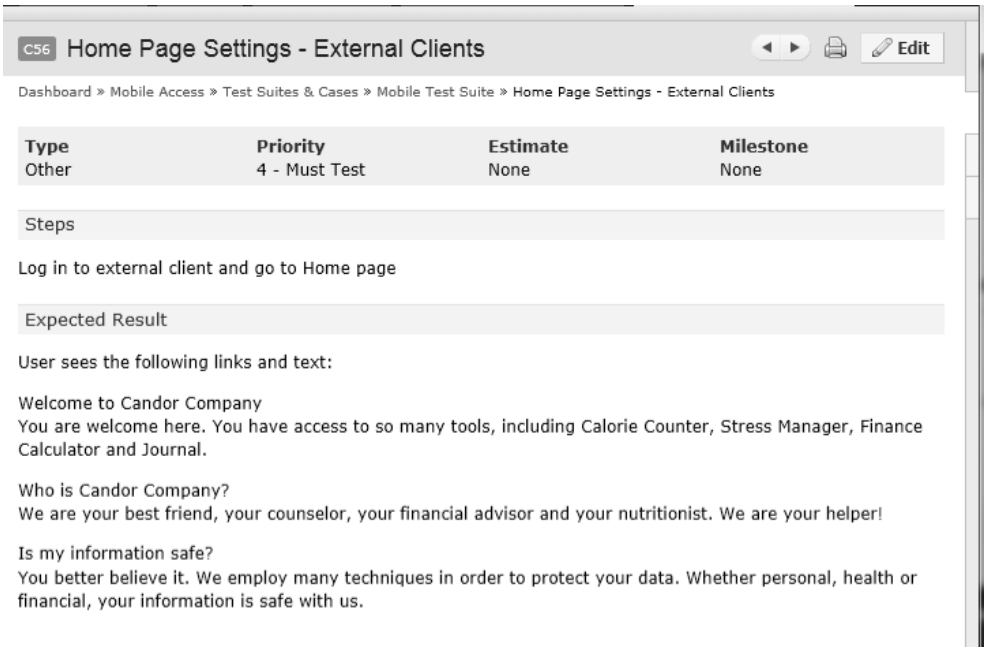
Up to this point, we've discussed the need to decrease the number of manual test cases in our regression test repository and purposefully maintain a low number throughout future development. In addition to limiting the number of tests, we also need to purposefully limit the *amount of detail* in the tests themselves.

Depending on the test, the format for a manual regression test can vary greatly. For that reason, I won't try to give you a standardized template for what a test case should look like. Instead, it's better to look at some guidelines and write your tests using these tips.

5.1 Include the intent of the content… not the exact text of it.

Often, a tester needs to verify the existence and intent of content but not necessarily the exact text. To keep your tests more readable and less fragile, record the main goal of the test and leave out the exact textual details. Not only will this make the test less likely to break, but it will also make the criteria for passing the test much more obvious to the tester.

In the two figures below, you will see two tests that are tough to read and are fragile due to the fact that there is exact text in the content:



C56 Home Page Settings - External Clients

Edit

Dashboard » Mobile Access » Test Suites & Cases » Mobile Test Suite » Home Page Settings - External Clients

Type	Priority	Estimate	Milestone
Other	4 - Must Test	None	None

Steps

Log in to external client and go to Home page

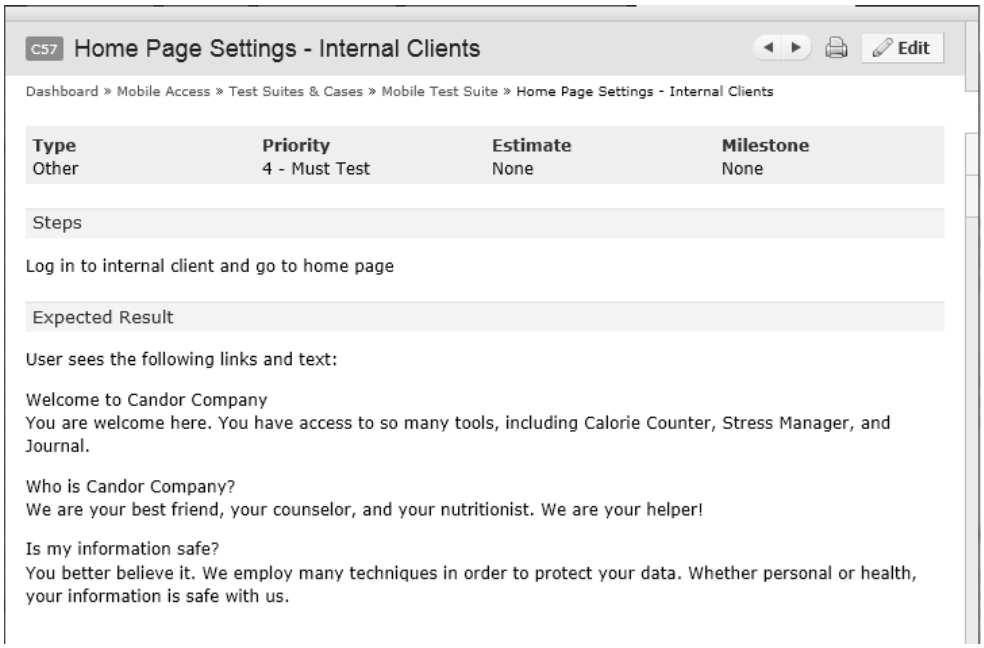
Expected Result

User sees the following links and text:

Welcome to Candor Company
You are welcome here. You have access to so many tools, including Calorie Counter, Stress Manager, Finance Calculator and Journal.

Who is Candor Company?
We are your best friend, your counselor, your financial advisor and your nutritionist. We are your helper!

Is my information safe?
You better believe it. We employ many techniques in order to protect your data. Whether personal, health or financial, your information is safe with us.



C57 Home Page Settings - Internal Clients

Edit

Dashboard » Mobile Access » Test Suites & Cases » Mobile Test Suite » Home Page Settings - Internal Clients

Type	Priority	Estimate	Milestone
Other	4 - Must Test	None	None

Steps

Log in to internal client and go to home page

Expected Result

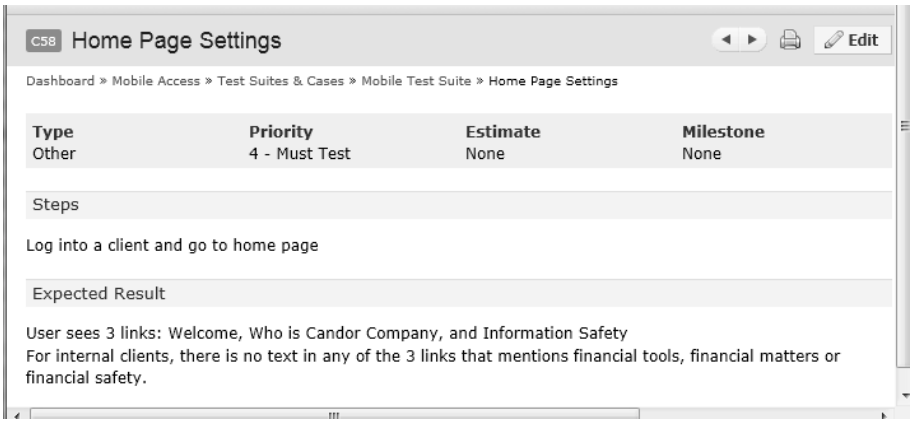
User sees the following links and text:

Welcome to Candor Company
You are welcome here. You have access to so many tools, including Calorie Counter, Stress Manager, and Journal.

Who is Candor Company?
We are your best friend, your counselor, and your nutritionist. We are your helper!

Is my information safe?
You better believe it. We employ many techniques in order to protect your data. Whether personal or health, your information is safe with us.

Using these guidelines, there shouldn't be precise text in these tests since it can frequently change in subtle ways, and we can indicate the intent of the tests much more clearly and effectively with the following test:



Of course, if the exact text is essential for legal reasons or specific client demands, then by all means, create a manual regression test that verifies that exact text. But usually, the main intent of a test is that the general concept or instruction is communicated. Write your tests in a way that ensures the user is testing for the general concept and don't make the tester sift through a paragraph of exact text to try to decipher the important elements.

5.2 Avoid writing extensive setup instructions or leading steps

In his blog, software tester and exploratory testing aficionado David Gilbert contended that "the amount of formatting and verbosity applied to a test plan is inversely proportional to the amount of actual good testing represented by such plan." I would have to agree. For most tests, I believe it is smart to assume that the person running the test has at least some knowledge of the product area they are testing. If they don't, they will ask questions that will enable them to start testing. It takes a lot less time to give a mini training session to your new testers than it takes to continually create and maintain specific instructions for every user flow. It simply isn't efficient to write your tests in a question-proof manner when most of the time, the people running the tests don't need that much detail to know what to do.

The figure below shows a test in a verbose format. This test is representative of many of the tests that are being written in organizations today.

Step Name	Description	Expected Result
Step 1 Monthly Journal link	1) Go to Child Health 2) Select [The Monthly Journal] on the homepage	The Monthly Journal page displays.
Step 2 Enter Journal link	1) Click [Enter Journal] on the left-hand side of the page	The monthly timeline page displays.
Step 3 Select Month	1) Select month 1 on the monthly timeline	The Create a Month 1 Journal page displays
Step 4 Enter Memories	1) Enter information to complete the [For your Baby Book Memories] section: babytest seems to smile the most when... testing babytest's favorite song seems to be... testing songs When I first held babytest, I felt... like testing The best baby gift I got was a... a new test gift babytest looks most like... daddytest The most memorable thing this month was when baby... tested A top news headline this month is: testing is fun	Information is entered with no errors
Step 5 Minor Health Problems	In the [Minor Health Problems] section, select Yes to a few of the conditions (e.g. colic, cradle cap, diaper rash...)	Radio buttons work with no errors
Step 6 Safety Precautions	In the [Safety Precautions] section, select No to a few of the precautions (e.g. use car seat in back, baby sleeps on back or side)	Radio buttons work with no errors
Step 7 Additional Things	1) In the [List up to five additional things that you would like to remember about Month 1] section, enter the a few things in text boxes. (e.g. Crying, Not sleeping...)	Text is entered with no errors
Step 8 Edit baby Journal	The Monthly Journal > Month 1 > edit 1) Click at [edit] button to make changes to baby	New information is saved

To make matters worse, in the company where I found this test, it was combined with six other tests that looked very similar.

I was able to combine all seven tests into one succinct test that does a better job of detailing what we are truly interested in regression testing. You can see this test in the figure below. By writing this test more concisely, it is easier to read, less fragile and helps the tester verify the behavior that truly needs to be verified.

	Step Name	Description	Expected Result
	Creating Child Account	Able to create child account from Family Members or from Living Healthy --> Child Health --> Create a Child's Account	
	Capabilities in Child Account	From within a child account, user can: - create a calendar event with reminder - enter a journal entry - add/remove/edit Child Health Record entries - enter Child specific tracker values - Click hyperlinks for content - print Health Record Reports and Emergency Card	
	CDC chart	CDC chart is displayed in graph section of Child Weight and Child Height trackers.	

If more details are necessary, then it is acceptable to include details. But it's important to make a conscious effort to decrease the verbosity of tests and keep them readable and maintainable.

5.3 Avoid excessive use of copy and paste

Copy and Paste has its place in our typing lives. It's a great invention and helpful when used in moderation. But Copy/Paste is also one of the major sources of confusing verbosity and inaccuracy in documented test cases.

Just as copying code and duplicating it in another part of your code base is a bad idea, the same is true for writing tests. It leads to many inaccuracies when test writers move too fast and don't make the small changes in each of the copied tests. Future updates to the tests also get made in one location but aren't propagated to all copied locations, leading to further inaccuracy. As for running the tests, consumers start to skim through the repeated details, assuming everything is the same as the previous test, and they miss necessary details.

If you find yourself using Copy and Paste frequently while writing your tests, it's a good sign that there is a better way to write the test. Consolidate all the repetitive details and only highlight where you expect to see differences in behavior. In the end, you have a test where things don't need to be repeated over and over again. After all, if you don't want to type something over and over again, what makes you think someone wants to wade through the muck to try to find the important differences between tests?

5.4 Only include details if a difference in behavior would result in a bug

How many times has a test resulted in behavior that ALMOST matches the test case, but doesn't EXACTLY match? What do you do? In many cases, the tester knows enough about the recent product development and is able to decipher whether the discrepancy is a bug or not. He often finds himself saying, "Well, it's not behaving *exactly* like what is documented, but it's following the intent of the test so I believe the behavior is correct." When this happens, then it is a sign that unimportant details have crept into the test case and muddied the waters. When writing or maintaining tests, only add details in the test case that would actually cause someone to log a bug against the test. This level of detail may vary from organization to organization, but your goal should be to make the tests as robust and low-maintenance as they can be without compromising their ability to verify the desired behavior.

6 How do I find time to clean up my existing test plans?

In order to be able to run every single manual test in every major round of regression testing, most QA organizations will need to do some serious clean up their existing and bloated test plans. Even if teams start writing good tests now, in order to be able to follow the new rule of manually running every test with every regression testing session, it is necessary to clean up the technical debt from years past.

Don't worry, there is no need to fret. This cleanup won't require months of re-writing of tests. The most important thing to remember is that your job during cleanup is not to rewrite every single test that has ever been documented for the product area. Your job is to glance through the hundreds of tests and pluck out the nuggets of wisdom worthy of being propagated to regression test status. Test plan by test plan, grab what you need and then delete the rest.

Throughout the cleanup of one my product areas, there were multiple test plans that had 100-200 tests in each of them. You'd think that grooming these would have been an insurmountable task requiring a large amount of resources for the effort, but it wasn't. By following these steps, I pared the test plan down to what was really necessary.

6.1 Get rid of the minor details

On the utest.com testing blog, one participant stated that "discovering the unexpected is more important than confirming the known." I agree. So the first step I took was to get rid of all the tests that just served the purpose of confirming a bunch of known details. I read every test and deciphered what the intent of the test was (the "nugget of wisdom"). For most of the documented tests, the small pieces of functionality weren't significant enough (or likely enough to change) to warrant their own regression tests. So I deleted them.

One thing to remember: As you delete test cases, you are not deleting *quality*. Most of these tests are inaccurate and haven't been referenced in years. You are increasing quality by grooming the test plan to something that is manageable and digestible.

6.2 Remove duplicate text

Throughout the same test review, I found dozens of situations where multiple tests were almost exactly the same, but had one or two minor differences. I made a note of those minor differences and deleted the multiple instances of the duplicate text.

6.3 Rewrite what is left of the test case

I then rewrote the test in a way that retained the important pieces of what I had found… and then referred to the various options/scenarios that should be tested if there was more than one pertinent scenario. Using the tips for section 5, this rewriting of test cases took a surprisingly short amount of time.

Remember, these are regression tests for existing functionality, not acceptance tests for new functionality. It isn't essential to cover every limit, edge case and use case. You simply need enough detail in the test to check that nothing has broken code that was previously working as tested.

6.4 Ease into delete mode

After years of writing test after test, you may be uncomfortable hitting that delete button. I understand. It takes time to retrain your mind after so many years of equating quality with the number of tests. So in the beginning, you can start out with a safety net. Create a folder or repository named "Deprecated". Then move the tests you plan to delete into this folder so you can refer to it in the future if you need to. After a few months of ignoring this folder and never needing it, you'll feel confident in trashing it for good.

Overall, the cleanup of some of the most bloated, unreadable, and inaccurate test plans took a matter of hours. The return on investment for these few hours of effort has been more than worth it release after release.

7 How can I break my old habits?

I recognize that many QA professionals have been over-documenting and over-complicating their test plans for many years. Even while attempting to adopt the new guidelines in this paper, old habits will continue to take you back down the path of excessive regression test writing. There are a few things that can help you continue down the proper path

7.1 Run a report at the end of a release

One of the most concrete ways to check on how well you're maintaining the size and complexity of your manual test plans is by running a report of how many manual tests didn't get executed during a release. In order for this process to work, the number really should be zero. If it isn't, you have two choices: Allocate more time to regression testing or Delete some of the tests that weren't important enough to run.

7.2 Do frequent peer review

Peer review has become a useful tool in many areas of software development. Whether it's two developers coding in tandem or a developer reviewing code changes before a development effort is considered done, there is great value in having multiple eyes look at code changes. At WebMD, we have begun to implement the same technique in the area of regression testing.

During a regularly scheduled meeting, the QA Analysts for a given product go over the stories that were completed since the last meeting. In these meetings, we aren't demoing the behavior itself (since that is done to the business and product owners during sprint demos). And this isn't a time for us to review every single functional, acceptance, and exploratory test we ran during our testing efforts. Instead, we briefly discuss the areas we tested (to ensure we didn't miss something) and then show the regression test(s) that were created or edited.

These short meetings help to keep us on track throughout the entire release. It prevents us from forgetting to enter any test cases… but it also keeps our desire to document in check. With peer review, we are able to discuss our test case writing and change course as we go, instead of when it's too late at the end of the release.

7.3 Keep on keeping on

In order to help QA professionals continue to make good decisions about what test cases to write and what to include in them, I have created some "bumper stickers" to serve as reminders. I encourage you to find the bumper sticker(s) that resonates with you. Print out a couple for yourself and post them on your office wall. Keep the list handy to refer to with your fellow QA professionals. Who knows? You may want to come up with your own to remind you of the ways in which you plan to shift the paradigm of manual test plan documentation.

Without further ado, here are the bumper stickers!



8 Conclusion

For the most part, QA professionals have good intentions when documenting tests. Yet those good intentions can often result in overgrown and useless test plans that no one can digest or maintain.

By being more purposeful and intentional about what we choose to document, we are much more likely to truly assure the quality of our products. Many professionals have been over-documenting manual test cases for years and the habits are hard to change. But with practice and peer review of test plans, we can decrease the technical debt in our overgrown plans and start to write tests that are maintainable for years to come.

Not every tip and instruction in this paper will be possible to implement in every organization. After studying it, however, each development organization will be equipped to analyze their manual regression test needs and make wise decisions based on those needs.

When this happens, we can reach a point where our QA professionals are doing more testing than talking about testing, and the quality of our products will greatly improve. Good luck as you change the way your organization writes and maintains manual tests. May you start reaping the rewards of those changes!

References

UTest Software Testing Blog. "16 Great Quotes For Software Testers" Mike Brown, http://blog.utest.com/16-great-quotes-for-software-testers/2011/01/.

Test Explorations. "The Theorem of the Well Formed Test Plan" David Gilbert, http://testexplorer.com/blog/?p=96

Get the Balance Right: Acceptance Test Driven Development, GUI Automation and Exploratory Testing

Michael Larsen
mkltesthead@gmail.com

Abstract

Depending on who you talk to, all of our testing challenges can be solved by doing some particular variant or flavor of “in vogue” software testing techniques. In the Agile community, Test Driven Development and Acceptance Test Driven Development are quite hot properties. So hot, in fact that some programmers and programming managers have declared that "Test is Dead" or, at least, is figuratively so. Others speak to the proliferation of frameworks and tools that allow for automated testing of the front end/GUI interface, and that this spells the future of testing. Others say that "active, sapient and human” exploratory testing is the truly effective method of performing software testing. Which group is right?

The answer is "all of them are right, and none of them are right". All three of these approaches, applied with maximum effort and efficiency as standalone initiatives, will not guarantee bug free code. Taken together, with a good understanding of where each area excels, where each area has deficiencies, and where each can leverage the strengths of each other, programmers and testers can bring a balance to development and testing efforts that are much more likely to find the issues that matter.

Biography

Michael Larsen is a senior tester with SideReel.com, a wholly owned subsidiary of Rovi Corporation, currently working at the Rovi site in San Francisco, California. Over the past two decades, he has been involved in software testing for products ranging from networking equipment to capacitance touch devices to Internet applications. Michael serves as a Director for the Association for Software Testing (AST) and is the Chair of the Education Special Interest Group. He actively teaches software testing through the Black Box Software Testing series of classes offered by AST. He is a black belt in the Miagi-do School of Software Testing, is the co-founder and primary facilitator for Weekend Testing Americas, and is the senior producer (and frequent commentator) for Software Test Professional’s “This Week in Software Testing” podcast.

Michael writes the TESTHEAD blog at http://mkltesthead.com/ and can be found on Twitter at @mkltesthead.

Copyright Michael Larsen 2012

1 Introduction

It's been a little over ten years since the Extreme Programming (XP) community, and other parties interested in improving the craft of software development, came together in Salt Lake City and developed what would become the Agile Manifesto. There is no question that a seismic shift in the way that software is being written, tested, and delivered has come about because of that meeting. Many new terms have been added to our software development lexicon because of it. Pairing of developers, testers, and combinations of either are now common in software development teams.

The notion of a dedicated testing team as a standalone entity is changing, with self-organizing teams covering many disciplines that were, and still are, treated as separate in traditional development environments. With these changes, however, a philosophical debate has emerged. By embracing Test Driven Development (TDD), programmers write tests first to make sure that the code is able to meet the call of what is being designed. Programmers then make sure code, as it is written, continues to pass existing tests. New tests are created to define and inform new functionality. Extending this model to incorporate the business goals, as well as defining them earlier, the concept of Acceptance Test Driven Development (ATDD) has gained popularity. We can look at ATDD as being synonymous with the Steven Covey rule to "begin with the end in mind", and making sure that the functionality being developed actually meets the criteria being requested by the organization [Gärtner, 2012].

We have seen an explosion in the availability and the capabilities of robust, effective and, in many cases, free software testing tools. Gone are the days when testing tools were only available from commercial organizations that charge exorbitant fees for licenses and proprietary standards that would only work within their applications. Languages like Java, .NET and Ruby have allowed the growth and popularity of frameworks like FitNesse, Selenium, and Watir. Natural language tools such as Cucumber structure tests into readable specifications that anyone in the organization can read and, theoretically, create. The ability to create tests to automate the front end and user interface of applications is more prevalent than ever.

Over the past several years, I have seen a number of industry figures look to these trends and say that, because of the developments of these tools and technologies, as well as using Continuous Integration (CI) to manage and deploy builds, dedicated testers aren't needed any longer. Testing, as has been said in a number of different mediums, is dead [Savioa, 2011]. To counter that, many voices have also come out to say that the ability to apply reason and dynamic thought to problems, to use "sapient skills" and active exploration, and that thinking, actively engaged testers are needed now more than ever [Tomlins, 2011].

This paper looks to explore all three of these areas, explore the strengths and weaknesses of each, and show that we need all of them to work together to be effective. Additionally, I will offer some suggestions as to how we can leverage all three areas to drive our testing efforts so that we can deliver what is most important, which is real value to our customers and stakeholders.

2 TDD, What Is It?

Test Driven Development (TDD) is an ongoing process, where a programmer works on small pieces of functionality, and in the process of writing the code, create tests that determine if the code in question is doing what it is supposed to do. Kent Beck wrote the book “Test-Driven Development by Example” in 2003, and in its preface, he sets out the primary goals of TDD. In short, the goal of TDD is to "create clean code that works" [Beck, 2003]. The process for doing that is the core focus of TDD.

To reach this goal, the programmer creates tests first that are meant to focus the expectations of the software. Each test, as it is written, is first written to guarantee a failure (after all, the code associated with these tests doesn't exist yet). Once the test fails, the programmer then works to create the functionality that will make the defined test pass. After doing this, the programmer(s) then examines the code they

have created and clean it up. Repetitive steps can be placed into functions or methods. This approach is encapsulated as "Don't Repeat Yourself" (DRY) and is formally called "refactoring". It is meant to make sure that the functions and procedures created answer the functionality needs, and that the code in question has a minimum of replication and duplication [Beck, 2003].

After cleaning up the code, the programmer runs the tests again, to make sure that the tests still work and that the cleanup/refactoring has not introduced new issues or caused the tests to fail. This process is used throughout the entire software development cycle. Each function, new feature, and new user story is driven by creating tests first, and then writing code that makes it possible for all cumulative tests to pass. If tests fail, then the process repeats itself until all tests pass (and all code, whether newly or previously written) passes all of the tests.

2.1 ATDD, how does it differ from TDD?

Acceptance Test Driven Development, while it may share many of the same words in its name, has a different focus from Test Driven Development. From One of Microsoft's teams in Israel, they describe the difference as “Unit tests (TDD) is about building the *Code Right*. FitNesse (ATDD) is about building the *Right Code*” [ArnonA, 2011].

TDD places the focus on functions, classes and methods. The goal is to make sure that the code that is written meets the specific goal of that piece of functionality, or for those given functions and methods, and that the resulting code is as clean as possible. By contrast, ATDD places the focus on User Stories. Stories are developed to explain the desired functionality, not from the perspective of functions and methods, but from the expectations of the customers. User stories can cover variations of functionality. It can mean anything from a new interaction screen, enhanced text, and improvements in performance, usability or other aspects that describe a feature as a user would see it.

TDD is used by programmers, and focuses on performing tests at the unit level (typically written in the same language as the application being written). ATDD is geared towards the members of the team that are not necessarily programmers. ATDD uses tools that allow a more natural language to present the requirements. The requirements scripts will look like typical manual tests, but they map to underlying automation code. This allows an automation framework to run our acceptance tests as actual executable steps. Various tool can be used (Cucumber, FitNesse, etc.) that allow testers to verify that, indeed, the programmers are building the Right Code.

Here’s a very basic scenario example, a user logging into Twitter:

```
Scenario: Twitter user signs in through Twitter connect button on sign in page
  Given I connect to twitter from the sign in page
  When I log into twitter with good credentials
  Then I go to the user private profile page
  And I can confirm my twitter credentials
```

The syntax above is Gherkin, which is coupled with Cucumber to allow the system to map the scenario statements to step definitions, which in turn map to actual commands in the programming language being used. In my environment, this would be Ruby, but Cucumber can interact with other languages and frameworks as well, including Java and .NET [De Florinier, Adzic, 2010]. We will see this same example get expanded when we discuss automating the Graphical User Interface (GUI) later.

2.2 Why are TDD and ATDD not Test Techniques?

For all of the talk about tests being a core part of both TDD and ATDD, we need to be clear; neither TDD nor ATDD are about testing. TDD and ATDD are design tools. They are used to help ensure that the software being developed meets the needs of the project. They help keep the project goals in focus. They help to ensure that the code developed does not get overly heavy or tries to accomplish too much. They help to ensure that "dead areas" of code do not develop, or can be removed. They arguably help make sure that programmers create more error free code. Every one of these is a win and a positive development, but they are not performing active testing.

What they are doing is repeatedly checking to make sure that the tests that are built into the system continue to work, and they help make sure that the code that has been written still meets the criteria of the design. What these tools do not do is perform active and dynamic testing of code, in the way that a focused, inquisitive, engaged and potentially devious human being can. Testing is the art and the act of asking questions of a product, and then developing and devising new and more inventive questions based on the answers we receive. Neither TDD nor ATDD do that. What's more, TDD and ATDD cannot test for questions developers never thought to ask.

2.3 Where do TDD and ATDD Excel?

Both TDD and ATDD are excellent ways to provide a disciplined approach to developing cleaner software. TDD and ATDD act as a brake on "cowboy code". It prevents software being developed that requires concentrated debugging days, weeks or even months after code was written. It also makes it possible for programmers who have not worked on a project to see the code in context with the tests that have been written.

TDD and ATDD both provide an advantage when it comes to creating Continuous Integration environments. Continuous Integration is the ability, whenever a new component or method has been committed to a project, to automatically run the tests associated with that module, rebuild the rest of the application, and run the tests associated with all of the features in the software application. These checks are performed repeatedly to ensure that new functionality works as the original design intended. Running these tests every time that the software changes keeps the code clean, as well as helps keep issues based on dependencies to a minimum.

Both TDD and ATDD are well adapted for Agile software development environments. They are well suited for short iteration cycles and the ability to work on small subsets of functionality in an iterative manner. Each directs the programmer to focus their attention performing "just enough design" and "just enough development". It eliminates the need for having to do all of the planning up front. The ability to focus on small areas of the code also has the benefit that refactoring can be done on a regular basis with each module as they are developed.

Another advantage is that the entire product team can handle the design and development work. The customer or their advocate also participates. TDD is typically done at the component level. ATDD allows for a broader range of technologies to be used. If we write our TDD code with Ruby, to perform ATDD, it is common to set up an environment where tools such as RSpec and Cucumber are utilized. Cucumber uses a language called Gherkin that allows anyone on the team to craft tests by creating Scenarios. Those scenarios and their statements are mapped to underlying commands. These underlying commands are written using Ruby, and they allow for the actual automated steps to be run. The big benefit from this is that teams can work together to develop tests. Even team members that do not have programming experience can understand the flow of tests and create their own scenarios.

2.4 What are TDD and ATDD's Deficiencies?

Both TDD and ATDD can give those who use them unrealistic expectations. The system of TDD and ATDD is not a magic bullet that will guarantee clean code that is bug free and without issues. Neither TDD nor ATDD replace good design or coding skills. The programmer is ultimately in control, and thus the framing will only be as good as the overall skill that the developer has [Tchepack, 2003]. There is also a tendency to over think or over apply these tools. When a change to a table format is needed for a web page, is it really essential to write a failing test first before making a change to the table structure?

Additionally, for many who are new to the ideas, the common approach is to "follow all of the rules" as they have been laid out and follow them all the time. Of course, context matters, and these tools are really only as good as the current context allows them to be.

The biggest challenge that is still left unanswered by TDD and ATDD is that, while they are focused on having tests run to determine that the code is being written the right way, and that the right code for the right purpose is being met, there is a limit to how much a handful of acceptance tests and unit tests can go in determining if the software being written is really going to behave well when a real human being starts interacting with it [Dalke, 2009]. While TDD and ATDD can tell you if an object exists, or if a page element is visible, it can't tell you if the load time is lagging, or if the order of elements appearing in the time frame it does will be frustrating to an individual user. It also doesn't say if the objects that are appearing are doing so in the correct context of the application, doing so at the right time, in the right place, when they are actually needed.

Additionally, TDD and ATDD do not answer aesthetic issues, those things that, while technically correct, actually require a human to determine if they appear in a way that is pleasing or in a way that is helpful.

3 What is Front End GUI Automation?

The Graphical User Interface (GUI) is how most people interact with computers and mobile devices today. Yes, there are some who are familiar with how to use the command line tools that come with various operating systems. The majority of activities, especially on the web and with mobile devices, are performed using the GUI. Unlike programs that start and stop from the command line, the GUI runs all the time, and programs are written to be always available and "waiting" for new input or direction.

To simulate the actions of users interacting with a GUI, there has been a proliferation of tools that help programmers and testers simulate the actions of users. This simulation activity is broadly referred to as Front End GUI Automation.

There are many reasons to do this type of automation. The most common is to follow standard workflows (logging in, navigating to pages, clicking on links, filling in forms, etc.). These can be as lightweight as a simple form filling tool (such as Texter) all the way up to full feature web page and application automation tools such as Selenium/WebDriver, FitNesse, TestComplete, and others.

3.1 How Does GUI Automation Differ From TDD/ATDD?

GUI automation is often used to help with constructing acceptance test cases and using those cases to check and demonstrate that the acceptance criteria has been met.

Expanding on the example mentioned in section 2, we use Cucumber to represent an acceptance test:

```
Scenario: Twitter user signs in through Twitter connect button on sign in page
  Given I connect to twitter from the sign in page
```

```
  When I log into twitter with good credentials
  Then I go to the user private profile page
  And I can confirm my twitter credentials
```

The Cucumber statements are an abstraction of what is actually happening in the code. Our GUI automation tool would then take these statements and map them to the appropriate tools that would make the statements work on the application or browser.

The below statements are groupings of selector statements. This technique is helpful when we want to group technical and specific statements so that they are not part of the actual acceptance test verbiage:

```
Given /^I connect to twitter from the sign in page$/ do
  step %Q|I am on the user sign in page|
  step %Q|I click "Connect your Twitter account" within ".login-container"|
end

Given /^I log into twitter with good credentials$/ do
  step %Q|I fill in "username_or_email" with "mytest497tw" within ".sign-in"|
  step %Q|I fill in "password" with "S4p3rl0g1n!" within ".sign-in"|
  step %Q|I click "Sign In" within ".buttons"|
end

Given /^I can confirm my twitter credentials$/ do
  step %Q|I should see "srtest101tw" within ".login"|
end
```

Each of these statements are mapped in a step definition to actual code that is run (in the examples below, the code is Ruby and we are utilizing Capybara to make the calls to the browser):

```
Given /^(?:|I )am on (.+)$/ do |page_name|
  visit path_to(page_name)
end

When /^(?:|I )click "(^")"(?: within "(^")")?$/ do |link, selector|
  with_scope(selector) do
    click_link(link)
  end
end

When /^(?:|I )fill in "(^")" with "(^")"(?: within "(^"*)")?$/ do |field,
value, selector|
   with_scope(selector) do
    fill_in(field, :with => value)
  end
end

When /^(?:|I )go to (.+)$/ do |page_name|
  visit path_to(page_name)
end

Then /^(?:|I )should see "(^")"(?: within "(^")")?$/ do |text, selector|
  with_scope(selector) do
    if page.respond_to? :should
      page.should have_content(text)
    else
      assert page.has_content?(text)
    end
  end
end
```

These are steps that allow the system to send commands to a browser and walk through them so that the

login process can be completed and then verify that the user is seeing elements that give them some satisfaction that the steps performed their job.

3.2 Where Does GUI Automation Excel?

GUI automation can be a tremendous blessing when dealing with repetitious steps that need to be performed for set up and take down of test environments. It's also helpful when certain states need to be created and prepared for testing. GUI automation can be a positive aspect and an important part of Acceptance Test verification and checking to see if the workflow as requested behaves appropriately.

GUI Automation also acts as a proxy for human interaction. For test steps that require a significant amount of interaction and verification, where components are loading and are visible in an application or on a web page, GUI automation helps the tester get to the point in the program where they can then examine interactions directly.

3.3 What is GUI Automation's Deficiencies?

As Corey Goldberg aptly describes, the GUI tends to be the most fragile layer of an application. It also tends to be the layer where a lot of automated test infrastructure is built [Goldberg, 2008].

There are a number of areas where GUI automation falls short of its intended goal. Some of this has to do with the way that testing tools have been marketed and are frequently used, especially by those who are inexperienced with test automation. There has been a great deal of rhetoric over the years regarding GUI Test Automation Tools and the way in which they are meant to be used. We have been promised test tools that provide record and playback simplicity. Suffice it to say that, in many cases, these tools have been oversold. While they do indeed record and playback, usually this creates overly specific scripts where the slightest change in the application will break the script. This creates a tremendous overhead of script maintenance or, often, the need to throw the scripts away entirely and start over again.

Even when the user can automate user interface interactions, there is a limit to how much can be realistically done. The basic steps for the acceptance test can be performed, following along the course of standard workflows and covering the steps that could be considered the "happy path", with some additional error handling for a few side cases. Automating all user scenarios, of course, would be out of the question, or even a small percentage of them. The return on investment would be low for the amount of work that would be required [Venkatakrishnan, 2009].

Another challenge that we face is that, while a tool will recognize if an element is on the page, it will likely not recognize if the context of that element has changed. An example; what if a user is checking to see if all of the links for a particular page are present? The test checks for the links, and if they are present, then the test passes. But what if, on that same page, the Cascading Style Sheet (CSS) is not loaded, and all of the styles for the page are, subsequently, not loaded? To the GUI automation program, the tests will pass, but a user would see the page and say "wait a minute, this doesn't look right!"

Also, If the appearance, location, text or description of the object changes, human beings have the ability to adapt to change much more quickly than we have the ability to reprogram the tests.

Finally, while front end GUI automation may be effective in helping to find issues the first time they are created, the odds are that these same tests will likely not find many, if any, issues beyond when they were first created.

4 What is Exploratory Testing?

Exploratory testing comes up in many discussions, but there seems to be confusion as to what it actually is.

One of my favorite definitions of exploratory testing is that it is “scientific thinking in real-time” [Bach, 2012] It's also the opportunity to put test design and test execution together. Exploratory testing is the opportunity to ask a program or application questions. Based on the answers we receive, we can then go any number of directions, and seek new or interesting answers based on the answers to those previous questions.

Exploratory testing is best described as simultaneous test design and test execution. To put it simply, the tests that we did before inform the tests that we will do now, and those tests will inform the tests we will perform later. Being able to have that flexibility and that opportunity to examine different paths is at the heart of exploratory testing. In addition, there are many implicit aspects of how we do things that, when exploited, fall into the realm of exploratory testing as well. Such things as varying the speed that we type, adding or reducing time to perform button clicks on the page, or getting more creative by opening up or closing down system resources so that our system has lots of memory or very little, lots of disk space or very little, or causing the CPU to run at a high percentage. Each of these can tell us interesting things about the software that we are testing.

Exploratory testing is not random, nor is it undefined. It does differ from "scripted testing" (of which automated testing is an example) in that the concepts are defined, but they are not completely pre-defined or run in a rigid sequence. Exploratory testing allows for a development of test ideas that are interesting, and again, are often based on the idea that they will be executed based on the information that previous tests provide answers for (or additionally having not provided answers).

Each time I run a series of tests, and if those tests are run all the time in the exact same way, then that is the epitome of scripted testing. The tests are re-run the same way, every time, without deviation. Suppose I take that same set of scripted tests (using my Cucumber test suite as an example), and I randomize the order in which the tests are run? If the tests are written where there are no contingencies on other tests, then there should be no difference. But often, I see that there is. Why? Because I discover that there are areas where dependencies or state conditions do exist. Randomizing the tests with the goal of seeing if we can uncover some unintended dependencies is an example of exploratory testing; we are conducting a controlled set of tests, but we are doing so in a way that asks different questions each time, and through the process, we evaluate the results to see if we are getting the expected clean runs, or if we learn something new from differing the approach.

4.1 What Makes Exploratory Testing Different From Automated Testing?

Automated testing, for the most part, is a sequence of steps that allows the tester to get to point in the program and run an assertion to see if a condition is met, or if an element is present on a page. It might be helpful if we were to change the terms used. Instead of saying "automated testing", I prefer to use the term "computer aided testing". By using those words as a framework, Exploratory Testing can, and often does, use elements of test automation to help accomplish its goals. The difference is that, instead of creating a mechanical process for running a sequence of steps, we instead look to get as much understanding from the application that we can, and that understanding can help us make further decisions about where we should aim our efforts. Scripted tests that do not deviate from their defined course do not give us that advantage.

Below are some examples of where automation excels and thing automation cannot do. I should point out that, even though automation cannot specifically do the things that are listed in the “Cannot” column, they can add to the ability of helping a tester make decisions based on the information they can provide.

Automation Can:	**Automation Cannot:**
Generate test data to be used in forms or as variable values.	Create curiosity.
Parse the output of a program and use it as input	Make sapient (actively thinking) decisions.

to another program.	
Create a log of actions and transactions.	Invent a new idea or an approach based on the output of a sequence of tests.
Alert if an assertion is met or not met.	Notice something unexpected unless we have already defined what is unexpected.
Search for patterns in output, or help to reveal patterns we did not know about.	Make a judgment call as to the value or importance of a piece of functionality.

Figure 1: A comparison of what we can and cannot do with automation [Bolton, 2010].

4.2 Where Does Exploratory Testing Excel?

Exploratory Testing allows the tester the ability to choose the sequence of steps that, at a given time and in a given context, can provide answers about the state or condition of a product. These tests allow for a very open and dynamic way of questioning an application. If we ask the same questions in the same way all the time, the odds of finding out something new and potentially interesting are much more limited. Imagination and creativity are what opens up the possibilities, as well as being able to determine which tests might be redundant or of lesser interest.

When I take my first look at an application or project, often there is little in the way of formalized documentation. Because of this, Exploratory Testing can be very helpful. Since I have had little interaction with a product at this stage, I am free to look at it and see if I can "figure out" what it should be doing.

Let's go back and look at our original acceptance test:

```
Scenario: Twitter user signs in through Twitter connect button on sign in page
  Given I connect to twitter from the sign in page
  When I log into twitter with good credentials
  Then I go to the user private profile page
  And I can confirm my twitter credentials
```

We have looked at how we could automate this process so that we can determine if the acceptance criteria is met. The acceptance criteria and the steps to meet it are straightforward. At the same time, we can also ask a number of different questions, many of which will fall outside of the actual story and acceptance criteria:

* Is there a login interface?
* Does it provide for proper error handling if I can't log in?
* Does it give me feedback to let me know that I have successfully logged in?
* Can I trick the system into letting me log in with improper credentials?
* Does it present me with information that could help me guess a login without actually having one?
* Are there ways that I can find out how to configure the application without explicit instructions?
* Can I figure out what a standard workflow with the application might be without explicit documentation explaining it?
* Where else could I use this functionality? Is it just on the login page, or can I find this control in other places? Do the designers even intend to have other places where I can access this control?

All of these examples, when applied, are where Exploratory Testing offers great strength. Each transaction could present me with something unexpected.

Exploratory testing is at the core of TDD and ATDD, at least as far as the initial understanding of what the programmer wants to do to implement the functionality. The idea of proposing a failing test first, and then creating code that meets the acceptance criteria, requires that the programmer give open consideration to what the application needs to do, and determine which of several avenues they will consider to implement the feature(s). Beyond initial development, the first time I create a series of automated test scripts, there

is a great deal of learning, fiddling around with the application, and walking down various paths to determine if I have a sufficient number of checks and assertions to create a robust series of steps to make for a meaningful test case.

Even with a number of checks and collective true or false statements, there is no substitution for the human brain to interpret the results and determine if they are meaningful or if they actually provide a real value to the development process. These checks allow us a jumping off point to look in other places, since the tedium of setting up the environment, or providing the data necessary to get us to a significant state, does not have to be manually run each time. One of my favorite tools to use when I am creating Cucumber scripts is to use a statement that is simply called "And let me see that":

```
And /^let me see that$/ do
  puts "PAUSED - Press Enter to continue: #{$1}\a\a\a"
  puts ""
  $stdin.gets
end
```

Programmers are familiar with this kind of a construct; it's referred to as a "break point" in code, where the program will run until it reaches that point, and then it will stop.

I use this often in my tests to help me make sure that I am able to get to a point that I am expecting to and determine if the conditions are right to perform additional steps, to ensure that elements are where they should be, but mostly it is used to give me a place where I can let the automated steps stop and hand control of the system to me. Frequently, this option allows me to then go and poke at any areas that might interest me. Again, automation cannot substitute for initial curiosity. It can only recreate the steps I tell it to do, and those steps are usually determined after I have had a chance to poke around and explore the system. Once I have done so, I can then go back to the script, hit the Enter key, and the script will pick up right where it left off, either to run the remaining steps in its defined sequence, or to tell me if the changes I have made or the areas I have explored will now produce an error because I have left the application in a state that the script does not know how to handle.

The key is that I have the ability to jump off and consider avenues that the script itself may not yet be coded for. Once I have added additional steps, then they become part of the series of pre-determined paths that I have created [Bach, 2006].

4.3 What are Exploratory Testing's Deficiencies?

The most direct deficiency with Exploratory Testing is also its greatest strength. Exploration is most important when treading on new ground. After I have gone and explored and mapped those areas, I am less likely to find new revelations by going over the same ground. This follows what Boris Beizer calls the "pesticide paradox"; by repeatedly running the same tests, we will not only not find new bugs, but the bugs that are left will be more resistant to the tests we do perform [Beizer, 1990]. Thus Exploratory Testing requires that we constantly be looking for new ways to explore areas we have already been over before.

Another issue is that, because most of the steps I perform require that I be actively engaged in examining new paths, there is simply not enough time to go through all possibilities. Exploration is labor and time intensive. There is a tradeoff; freedom to explore vs. time to perform that exploration. The answer then frequently becomes "why not just automate the steps I already know about"? While this will help run many tests faster, it also restricts the avenues I can drive the application.

Perhaps the biggest challenge with Exploratory Testing is the fact that sentient humans do it. Sentient humans also have a challenge that computers do not have; sentient humans get bored. If we consistently run through a variety of exploratory tests, over time, we may naturally start to fall into a routine. While we

tend to think that we are exploring, we are actually treading the same ground, doing the same steps we always have. We start to do things by rote, even unconsciously [Bolton, 2010].

5 Can We Create a Balance?

Every team and every product is a little bit different. As I'm sure you have seen from the examples given previously, none of these steps will stand on their own as a be all and end all for delivering quality software. They are all interdependent, and to speak of doing Test Driven Development, Acceptance Test Driven Development, Front End GUI Automation and Exploratory Testing as separate activities is simply not true. TDD and ATDD, as well as front end GUI automation, are activities that we perform. Exploratory Testing is a mindset and an approach that we use when we perform those activities.

Exploratory Testing can be as manual or as automated as we choose to make it. Test Driven Development and Acceptance Test Driven Development are, by their very nature, exploratory endeavors, at least at first. Developing and defining the acceptance criteria for stories that drive development are likewise exploratory. Front End GUI Automation starts from the desire to capture and use the discoveries we make when we are examining the requirements and how to implement them. In all of these steps, testing is at the center of the activities being performed, and testers very much have a role in them being performed.

When the "testing is dead" conversations started in 2011, it was not to say that there would be no more testing, it was to say that testing needed to be a part of an entire process of quality and improvement, starting at the earliest stages of design and development. The idea of testing as a standalone, separate, after the fact endeavor where a group of people would take a product and shape it into quality, separate from its initial development and design, is indeed dead (or dying) *and that paradigm deserves to die*. Testers cannot "bake in" quality after the fact. We can identify problems and report on them. We can explain issues with design and requirements, but as a standalone group we cannot shape the quality of a product by ourselves. Agile teams over the past decade have come to this realization, and the idea of testing as an end of the line process is largely history for these teams. Instead, exploration, automation, and testing all come much earlier in the process, and involve not just testers, but everyone on the product team.

With that, here are some ways that we can leverage the "testing everywhere" idea, and help develop a balance for all of these goals, and see them as interdependent and not as separate and standalone activities.

5.1 TDD and ATDD are Exploratory

From the first time a story is presented and criterion is established, a development team explores the options available to them. The customers provide us with their goal and their hopes for what the software will do. Developers and testers weigh in on what these requirements mean and how they will be implemented. Rather than testing being seen as an after-effect, it is part of the initial development effort. Rather than saying "here's what we will do and, oh yeah, we will test this all after we finish", testing comes first. By putting the emphasis on testing and making sure that the piece of functionality under development is designed with testing in mind, code can be written to make sure that it meets the criteria it is being designed for.

Requirements do not develop in a vacuum. Instead, they are discussed and the respective stories are fleshed out by examining and understanding the goals of the customer. Testers have an opportunity at this early stage to consider the customer's requirements and offer input on ways that the product can be tested. While programmers consider tests at a unit level for the modules they are programming, testers can also help guide and provide consideration for testability in areas the programmers might not initially consider. This process takes into account different factors depending on the context in which the new code is to operate. Exploration is vital to this process. In many cases, functionality is vague and amorphous early on. Exploring the approaches, asking a lot of "what if" questions, and actually thinking

out the ways that the software needs to work, how it might fail, and how we can effectively and helpfully alert the user to errors, is paramount to its success [Pier and Kaner, 2012].

5.2 Focus on Delivering Business Value

As a tester on a development team, we need to perform many tasks. While finding issues or examining acceptance criteria is an important aspect of this, the key area we need to focus on needs to be "is what I am doing helping to deliver real value to my customers"? To that end, does it make sense to do a lot of up-front automation for requirements that may or may not stay intact? Do we want to focus on the business value and how the requirements or goals of our customers can be met? If we put the business value first, that will guide us in our testing efforts and the approaches that we use [Crispin, 2008].

5.3 Automation Does Not Have to Be Permanent

The unit tests that are created to help guide TDD and ATDD are part of the process of writing software, and subsequently, those tests are part of the code base early on. Later automation efforts, such as those that lead to developing integration and regression tests, may have several iterations. Some automation will make sense early on but not be of much help later in the process. Not all automation is equal, and not all automation should be approached as though it will be a permanent part of the testing process. There needs to be room for "throwaway automation", steps that can be used to help explore different avenues.

Consider the difference between freight or passenger trains and taxi cabs. Freight and passenger trains can move a lot of people and cargo, but they are limited to the rails the trains can ride on. In this case, the rails are a permanent infrastructure, much like many of our dedicated and most important automated tests. By contrast, a taxi cab can be called to appear anywhere and take us anywhere (granted, for a price) and then we can go off and explore as we see fit. Likewise, we can use less permanent automation to do the same thing when we test. Like a taxi cab, we can use this temporary automation to take us to points of interest in the software, and then jump off to follow leads and see where they go. Some of these will be dead ends, and some will help identify key areas and workflows that are important to the success of a project. Be willing to experiment and see which steps will work and get you to where you need to be.

5.4 Exploration is a Mindset

As stated previously, Exploratory Testing does not live outside of automated testing. For many, Exploratory Testing is seen as being a manual process, one in which a tester is actively engaged in looking at the system and considering where to go with each question that they ask. The questioning aspect and the ability to adapt to the information gleaned from the questions asked sits at the core of Exploratory Testing. Automation and Computer Aided Testing can be, and often is, central to getting to the point to successfully explore those avenues.

Much as paddling upstream to explore a river's tributaries requires a well-stocked canoe (and possibly a team of rowers), having a team of automated tasks that allow a tester to get where they want to go inside of an application, and then allowing them to step off and look around, can be tremendously helpful. Again, Exploratory Testing is not a technique. It is a way of thinking about questions and answers. It is a dynamic exchange, with a give and take based on the information received. The approach can be entirely manual, or it can have many automated steps. The key is to know how to get where you want to go to look deeper.

5.5 Utilize Personas to Help Visualize Your Goals

One of the most effective tools I have found when it comes to exploration of an application or a feature is to put myself into the shoes of as many potential customers of our product as I can. At SideReel, that means understanding the interests, desires and ways of interaction of many different groups of people. The interests of a 16-22 year old girl, and the ways that they interact with the devices they use to discover

and watch television show and learn about them, are different than the ways middle-aged men might. Also, the shows and the content that appeals to one group often do not appeal to another group.

Relying too heavily on one group of customers may help see what they like or desire in a product, but may also mask what others would like to see. Using personas helps us get inside of these customers word views, and gives us a good understanding of what they effectively want to do and see. Personas are a key ingredient to effective exploration. Not only to they help us frame our questions, they also help us develop entirely new questions based on a variety of contexts.

5.6 Use Session Based Techniques to Keep Things Fresh

Testing and having 100% regression coverage is not something that is going to be achieved overnight. It takes time to examine requirements, develop robust tests that are not flaky, and getting all of the tests to behave in a way that is independent of each other takes time. Also, there is a tendency to grow weary when addressing too many things at once. Developing small charters and executing on them in set periods of time can help focus testing efforts on areas that are most important, and allow for a variety of approaches to examining the product. By keeping our focus on a small sub system, or by using a particular persona, and doing so within a specific context, we can keep our testing fresh, our eyes and minds focused, and we can work on what's most important, which is delivering the best product we can to our customers.

6 Conclusion

For many organizations that have adopted Agile and Test Driven techniques, testing is very much alive and well. It happens at every level of product development and delivery. For many, a high amount of automation exists, and for some, that automation is developing as the project matures. The avenues of exploration are many, and the possible questions to ask are as limitless as the imaginations of the team members that are working on a project. Quality is no longer the end tasks of a group of testers whose mission is to pound quality into a product, it's an integrated series of processes that are owned by and practiced by the entire team.

Test Driven Development focuses on writing correct and clean code. Acceptance Test Driven Development focuses on making sure the functionality being delivered makes good on the goals and promises made to our customers, and that the value they are expecting to receive is present. Automation happens at many levels and in many interactions, at the unit, system and user facing levels. Some tests will be end to end and require little human interactions. Other will be a way to get to a certain destination and allow tester the ability to jump off and have a look around interesting areas. Exploration happens at all of these levels, and is an approach and a mindset, not a methodology. For software to be written in a way that is effective, meets the needs of customers, and has a high degree of quality, all of these components need to work together and be considered.

References

Gärtner, M. (2012) “Acceptance Test Driven Development: By Example”, Addison-Wesley

Savioa, A., (2011), “Test is Dead”, http://www.youtube.com/watch?v=X1jWe5rOu3g, Electronically retrieved on July 24, 2012

Tomlins, M., 2011, "Test Is Dead, Long Live Test", http://mtomlins.blogspot.com/2011/12/test-is-dead-they-say-long-live-test.html, Electronically retrieved on July 24, 2012

Beck, K. (2003). “Test-Driven Development: By Example”, Addison-Wesley.

ArnonA, 2011, "Comparing TDD With ATDD", http://blogs.microsoft.co.il/blogs/arnona/archive/2011/02/14/comparing-tdd-with-atdd.aspx, Electronically retrieved on July 24, 2012

Tchepak, D., 2011, "Why TDD is Hard and What To Do", http://davesquared.net/2011/03/why-learning-tdd-is-hard-and-what-to-do.html, Electronically retrieved on July 24, 2012

Dalke, A., 2009, "Problems With TDD", http://www.dalkescientific.com/writings/diary/archive/2009/12/29/problems_with_tdd.html, Electronically retrieved on July 24, 2012

De Florinier, D., Adzic, G., 2010, “The Secret Ninja Cucumber Scrolls”, http://cuke4ninja.com/download-pdf.php, Electronically retrieved on July 24, 2012

Goldberg, C., 2008, "GUI Automation Considered Harmful", http://coreygoldberg.blogspot.com/2008/09/gui-automated-considered-harmful.html, Electronically retrieved on July 24, 2012

Venkatakrishnan, S., "Problems with GUI Automation Testing", http://developer-in-test.blogspot.com/2008/09/problems-with-gui-automation-testing.html, Electronically retrieved on July 24, 2012

Bach, J., "What is E.T.?", http://www.satisfice.com/articles/what_is_et.shtml, Electronically retrieved on July 24, 2012

Bolton, M., 2010, "Can Exploratory Testing be Automated?" http://www.developsense.com/blog/2010/09/can-exploratory-testing-be-automated/, Electronically retrieved on July 24, 2012

Bach, J., 2006, “Manual Tests cannot Be Automated”, http://www.satisfice.com/blog/archives/58, Electronically retrieved on July 24, 2012

Beizer, B. “Software Testing Techniques”, Second edition. 1990, Intl Thomson Computer Press

Bolton, M, 2010, “Exploratory Testing and Interviews”, http://www.developsense.com/blog/2010/04/exploratory-testing-and-interviews/, Electronically retrieved on July 24, 2012

Crispin, L., 2009, "Beautiful Testing", O'Reilly Press, Excerpt from http://lisacrispin.com/downloads/Beautiful_Testing_ch15.pdf, Electronically retrieved on July 24, 2012

Pier, K., Kaner, C., 2012, "Episode #105: Exploratory vs Automation, Part 1", This Week in Software Testing, Software Test Professionals, http://www.softwaretestpro.com/Item/5604/TWiST-105---Exploratory-vs-Automation-Part-I/podcast, Electronically retrieved on July 24, 2012

Automated testing: a key factor for success in video game development. Case study and lessons learned.

Christian Buhl, Fazeel Gareeboo
cbuhl@ea.com, fgareeboo@ea.com

Abstract

This paper outlines how the use of automated testing was a key factor in significantly improving the development of a game. We describe our game development process, how we implemented automated testing, and the results of our implementation on key metrics.

Biography

Christian Buhl is currently the Technical Director for the NCAA Football franchise and for the Core Football stability team at Electronic Arts in Orlando, Florida. Christian has a B.S in Computer Engineering from the University of Central Florida and did postgraduate work in Modeling and Simulations at the Naval Postgraduate School. Prior to working at EA Sports, Christian was Lead Engineer for America's Army, the official video game of the United States Army.

Fazeel Gareeboo is a development director at Electronic Arts in Orlando, FL. He currently manages the Development and Release Engineering team, which includes the automated testing group.
Fazeel has a B.Sc. in Computer Engineering from Manchester University and an MBA from Nottingham Business School.

Note on references to colors in this article
Most of the charts in this article have been converted from color to black and white to abide with the printing requirements for the proceedings. The main colors of note in the chart are red and green, and red converts to a darker grey than green.

1 Introduction

EA's Orlando studio (known as Tiburon) develops four major sports video games for Electronic Arts, and this paper outlines how the use of automated testing was a key factor in significantly improving the development of one of those games.

2 Game Development process

The regular development process for our iterative sports titles follows a sequence outlined in the diagram below:



Figure 1 - Game Development Process

The main stages of the game development process that this paper deals with are:

- Full production
- Post production

Full production is defined as complete when QA approves the Alpha build (i.e. the game is feature complete).

2.1 Alpha

Post production comprises of the following phases:

- Alpha
- Beta
- Final

Alpha is typically the most problematic phase, as this is when QA assigns significant resources to testing the game and the game team is in bug fixing mode. Before Alpha, QA staff assigned to the game is usually just a few people, and their role is to help the game team test and tune the game and verify that the game is feature complete. Getting to Beta requires the bug count to be down to 0 non-shippable bugs (crashes, hard-locks, etc).

The constraints of project management are often described as the Project Management Triangle, with Cost, Time, and Scope. In sports video games, the schedule is tied to the season for the real sport, and so there is very little flexibility with Time.

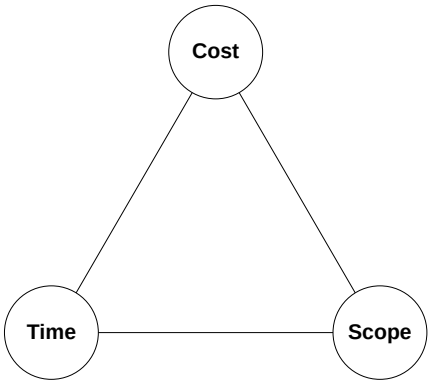


Figure 2 - Project management triangle

Once the game enters Alpha, significant resources have already been invested in feature decisions, and marketing may have already begun, so it is difficult (but not impossible) to reduce Scope. This leaves Cost (developer resources) as the only constraint that can be easily controlled, leaving only a few options in order to hit our dates:

- Increase working hours of existing team (crunch time)
- Add more staff to team

The above adjustments are painful, and hence undesirable. We usually start off with having everyone work extra hours, and if sufficient progress is not being made, we add more staff to the team, and so on.

On previous iterations of the NCAA Football franchise, we typically had poor visibility into the stability of the different parts of the game, which naturally led to poor overall stability. For example, a mode that was not actively being worked on might not be visited for weeks, and when a developer finally needs to work in that mode, it might crash immediately, and the developer has no idea when or how it broke.

When a game enters Alpha with such poor stability, this leads to a high number of defects, and consequently a large increase in hours for the team. This can lead to staff burnout and turnover, which subsequently impacts the quality of the next title.

2.1.1 The search for a good Alpha

In the search for a 'good' Alpha phase, it was evident that we needed to focus on the full production phase, as this was when the defects are originally introduced into the game (Jones and Bonsignour 2012 p188). Specifically, we needed to add process to our Production that would reduce the number of defects that were in the game when we hit Alpha. The Project Manager for the NCAA Football franchise put it this way – "There is no better Alpha. Only better Production".

In NCAA Football 11, we introduced a Quality Assurance Certification (QuAC) process. This is where a developer can ask an embedded QA tester to test their build with local changes. This allows the changes to be validated before checking into the main code repository. Major defects found through QuAC prevent the developer from submitting their changes, and minor defects can be flagged for immediate follow-up. This approach worked well, but takes time (need to push local build to QA tester's machine,

then wait for test result) and is not scalable (we had ~ 5:1 ratio of software developers to embedded QA staff).

QuAC could only get us so far. We had to look for a more scalable solution, and so we focused on improving our use of automated testing.

3 Commit to Stability

3.1 Stop the line

One of the key drivers for focusing automated testing was the NCAA Project Manager, who wanted to implement best of breed test automation. He was influenced by the 'stop the line' literature from Toyota manufacturing (Liker, Hoseus, & Center for Quality People & Organization, 2008). One of the reasons why he was looking into this was regularly hearing the complaint 'Alpha is bad' and 'we are working too many hours' and a hunch that automated testing could be a solution to this problem.

3.2 Support from the Top

Before he pushed for implementing automated testing on the game, the Project Manager first pitched the concept to the Executive Producer, to get buy-in from above. Once this was achieved, he gave clear objectives to the development directors and lead engineers, and made it clear that investing time and resources into automation would be a priority. He also asked for a dashboard to be created that tracked the state of automation.

Once the system was up and running, the Project Manager checked the dashboard every morning for several weeks, and if anything was broken, he would go and ask why and when it was going to be fixed. This made it very clear to the entire team that stability and automation were a high priority from the leadership of the team.

3.3 Creating a Vision

The NCAA Team is divided into several smaller teams called pods. The Core Pod is responsible for non user-facing features and technology, such as memory, technology upgrades, and stability. The Lead Engineer and Development Director for the Core Pod were assigned the task of creating and implementing the policies to make automation work.

The Lead Engineer and Development Director approached stability as if it were a feature of the game. This meant they had to brainstorm it, define the expected outcomes, build a plan to get there, and implement that plan (see section 4 for details of the plan).

3.4 Investing the Resources

With the entire NCAA leadership team committed, the Core Pod was able to invest significant resources into maintaining stability and improving the stability process. Roughly half of the Core Pod's capacity (5 engineers) was spent on stability during the cycle. Additionally, we were able to leverage a large amount of work from the Central Test Engineering (CTE) group. This was a central group setup by the studio a few years ago to lay the groundwork for automated testing of our games.

3.5 Engaging the Team

Finally, one of the most important factors in the success of the stability plan was the way we engaged the team. The NCAA team has a weekly programmer meeting, and we (the Core Pod) used this as a launching pad for the stability process. We communicated the vision for stability to the engineers and

explained how the stability process would benefit them (better Alpha, improved workflow, prevented from being blocked).

Before any part of the process was being implemented, we communicated clearly in advance to the team what it would be, so that everyone knew what to expect. Furthermore, we constantly solicited feedback on what worked/what didn't, so that we could continuously iterate and improve the processes.

Finally, at every programmer meeting, we showed all of our stability dashboards to the team, and discussed any ongoing issues if necessary.

4 Automated testing and its challenges

4.1 Test and Measure It

We began with a pretty simple dashboard of green Checks and red X's. This was a good visual way for everyone on the team to tell what was broken and what was not.

Over time, we added a LOT of detail to the marks (orange triangles for initialization issues, '!' for script loss issues, red diamonds for assert failures), this was mostly useful for the stability team and engineers to help triage issues. In particular, the format of the data was very useful – with CLs on one axis and tests on the other - this way it was very easy to see that a "Script Lost" (screen flow) issue started in the "Dynasty" script in CL 123456. This made it pretty easy to look at the CLs in this range and look for things that were related to Dynasty.

Later on we added even more functionality around here. We have a separate dashboard now that shows asserts, and one that shows crashes.

4.1.1 Automated Testing Infrastructure

The first step to implementing automated testing was to create the infrastructure to support it. We first setup a system that:

- grabs every new game build
- copies it to a game console
- runs a series of test scripts on it
- reports the results to a dashboard

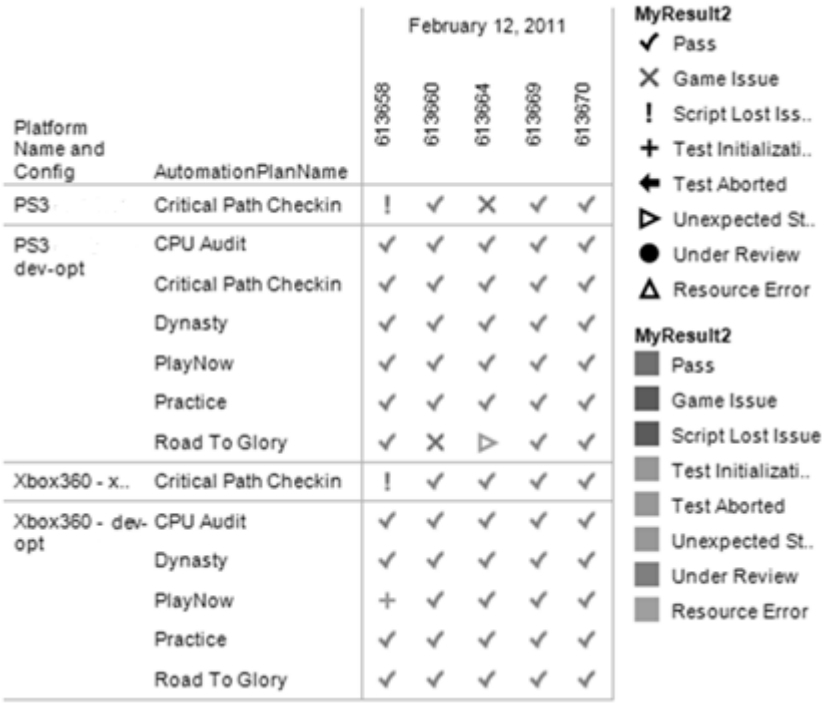


Figure 3 - Game Smoketest Dashboard

The smoketest dashboard shows the results of each script for each game build. The dashboard was accessible to the whole team, and was made to be very visual and color coded, so it was very easy to tell how stable the game was at any time. This took care of regularly testing the main build and reporting on the results.

4.1.2 Bots

We also had a system in place that ran bots every night. The bots are very simple AI scripts that can play games over and over, either solo or in online matches against each other. It tracks how many games completed successfully, how many failed, and detailed information on failures. The bot system:

- Grabs a game build every night
- Copies it to numerous game consoles
- Runs games continuously for a set period of time
- Reports the results to a dashboard.

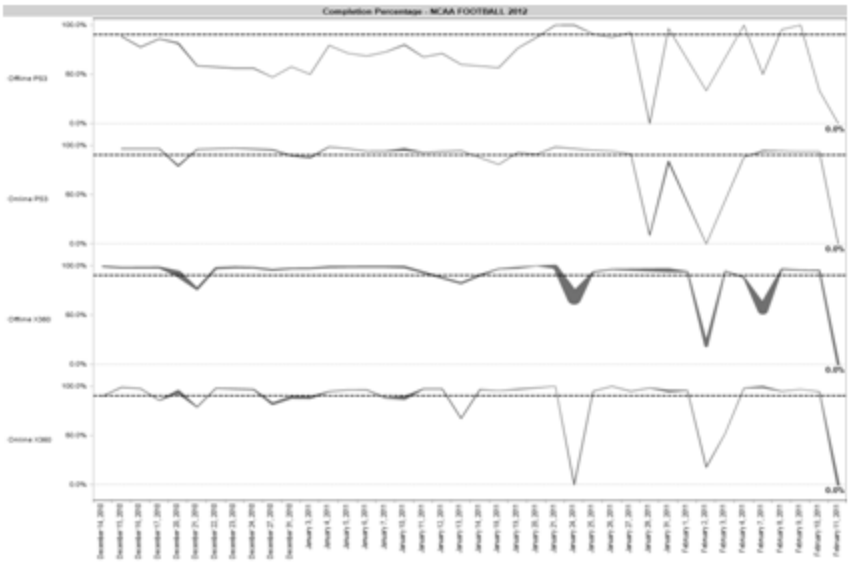


Figure 4 - Bot Dashboard

The bot dashboard shows the percentage of games that passed for each nightly run for each of the four build configurations that we were testing.

4.2 Targets

We set targets for both our smoketest and bot results. We set both a pass percentage target and a recovery time target for each one.

For automation, our target was 100% with a recovery time of 1 day. The expectation was that we would get at least one build every day that passed every single smoketest.

For bots, our target was 90% with a recovery time of 3 days. This longer recovery time was primarily because bots have long turnaround times. We only ran one full bot run per day. If there was a "deep" issue that caused a failure (for example, memory fragmentation after multiple games), it could take hours to triage and debug. It was simply not realistic to expect every bot issue to get resolved within a single run/a single day. Three days gave us three runs to try to get a fix in.

We later updated our dashboards to show this specific information.

The targets were set by the stability lead, who picked the shortest targets that he thought were reasonable.

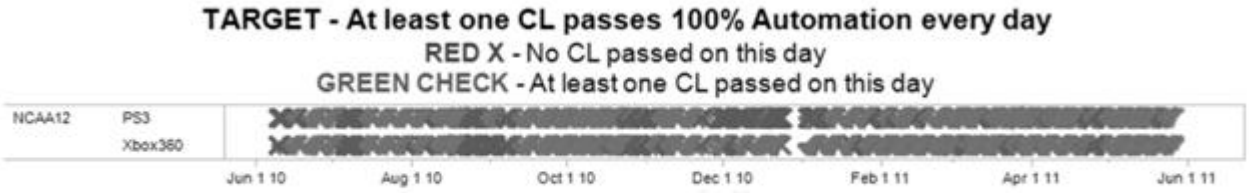
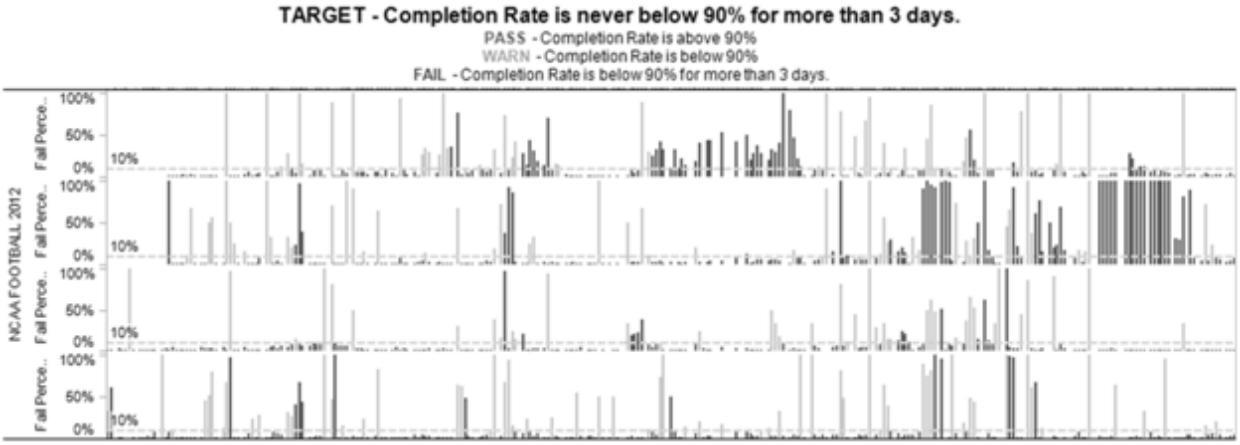


Figure 5 - Game Smoketest Targets



Game Bots Targets

4.3 Prevention

The test infrastructure ensured that we could test and measure the stability of the game. The next step was to make sure that we kept the game stable by preventing issues from being entered into a main build of the game.

4.3.1 At-Desk Automation

We provided every software engineer with the ability to run automated testing at their desk. This was integrated into the tool that software engineers used to build the game to make it easy for engineers to run. The process was simplified so it would take as little time as possible.

We created a "Critical Path" test script that went in and out of all of our game modes (but no deep testing), to ensure that no game mode would be broken. For example, we have a Dynasty mode where the user can control a college football team through multiple weeks and multiple seasons. The Critical Path script entered the "Dynasty" mode of our game, completed the creation stage of a Dynasty, and then immediately exited the Dynasty, without progressing through any weeks. This ensured that Dynasty was never broken at the highest level (creation), but didn't catch deep issues after multiple weeks.
All engineers were required to run *at least* the Critical Path script prior to every checkin. The Critical Path script was run on local consoles, and was kept fairly short (<10 mins) to ensure that it would be convenient for engineers to run.

There was no exact cutoff on what was "too long", this was mostly just a gut call that under 10 minutes was fast enough that it didn't interfere with workflow too much. In many cases, you could just start the script and when you came back to check it after looking at something else it would be done.

Interestingly, after adding this requirement, one of the most experienced engineers on the team, who had a reputation for never breaking things, mentioned that using the Critical Path script was a huge time savings for him. He had previously done a lot of the same testing manually before all of his check-ins.

Together with the 'test automation at desk' feature, we created a report that showed when the main build was broken, which engineers checked in code for that build, and did not run the critical path test at their desk before checking in (CheckInChecker Dashboard). This provided the game team the necessary tools to address 'destructive behavior'.

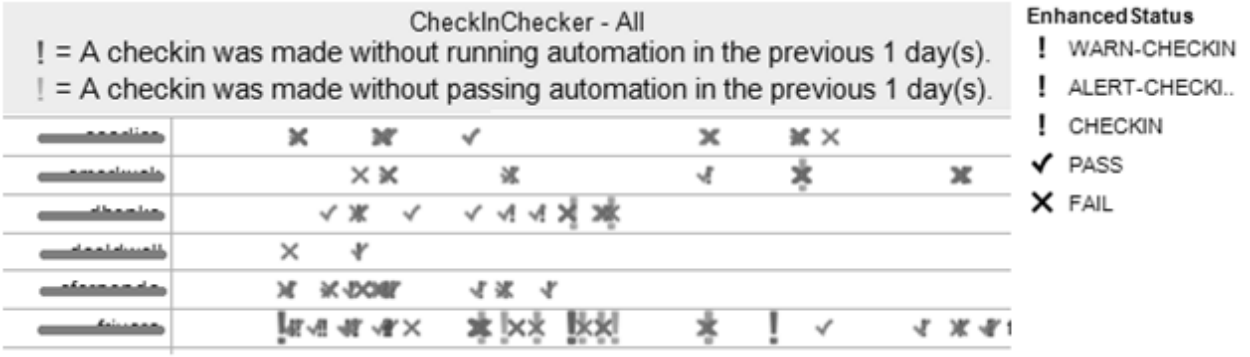


Figure 6 - CheckInChecker dashboard

Getting engineers to run automation before checking in also addressed one of the challenges of automated testing, which is that of making sure the test was in synch with the game. Previously, the test would sometimes fail because the game was changed and the test was failing to navigate the game – as opposed to detecting a failure in the game itself. With the new process, engineers would realize that the test needed to be updated because of their changes to the game, and they would check in the new test with their game code changes.

4.3.2 Expand Coverage

Initially, the Critical Path script was very shallow, it went in and immediately out of a single game, and then in and out of each game mode, without tracking asserts.

As we found things that have caused frequent stability concerns, we have added them to the Critical Path script, keeping in mind that we want to keep it fairly short. For example, our Critical Path did not initially complete a game; it just went in and quit the game immediately. However, we were seeing issues introduced during the End of Game state, so now the Critical Path simulates to the end of a game and exits the game normally.

Most notably, our Critical Path now tracks and fails when it hits an assert. This helps to reduce the introduction of code that is doing bad things, but doesn't actually crash (yet).

We have also added new scripts whenever we have added new game modes, and have expanded other scripts as well.

4.4 Reaction

Having automated testing on each build, and making it easy to run testing at desk, does not mean that the main build will never break. This meant we had to have a policy for reacting to build breakages.

4.4.1 Stop the Line

Our first and most important policy was that if our stability targets weren't met – if we did not get a 100% smoketest pass after a second day, or if we did not get a 90% bot pass after a fourth day, we would stop the line and freeze the depot except for stability fixes – no matter what. Even if freezing work put sprint deliverables at risk, we would do it anyway. The same policies applied to test code. If someone made a change to the Critical Path script itself that was broken, we would back it out. This happened many times, and was just treated the same way as any other issue that broke scripts.

The rationale for this was simple – meeting our stability target was a required deliverable for each sprint, so the sprint wasn't going to pass anyway if we didn't freeze. To reiterate, this obviously requires buy-in and commitment from the team leadership to be willing to put feature deliverables at risk to ensure that stability stays on track.

4.4.2 Back it Out

While the Stop the Line policy was and continues to be our most important policy, it obviously has a very high cost, as we are blocking the work of a large number of developers, particularly if the freeze lasts more than a day (which it usually does).

To mitigate this, we later added a "back-it-out" policy for any smoketest or bot breakage that we could identify. This meant that if a smoketest failed or if there was a significant bot issue, and we could reasonably identify the changelist responsible, we would immediately back out that change. The developer who checked it in would be responsible for resubmitting the change without the offending breakage.

This had the added benefit of impressing on the team the seriousness of game stability, and helped to ensure that all the engineers ran the tests before checking in their changes.

5 Results

The process described above was first tried on NCAA 12 (worked on during 2010-2011).

The results were very positive. Using our own measures of stability, we made demonstrable improvements over the previous year. In NCAA11, our Central Test Engineering team was actually running a small set of smoketests, but without much visibility. The following chart compares NCAA 11 to NCAA 12 across the entire cycle for both titles:

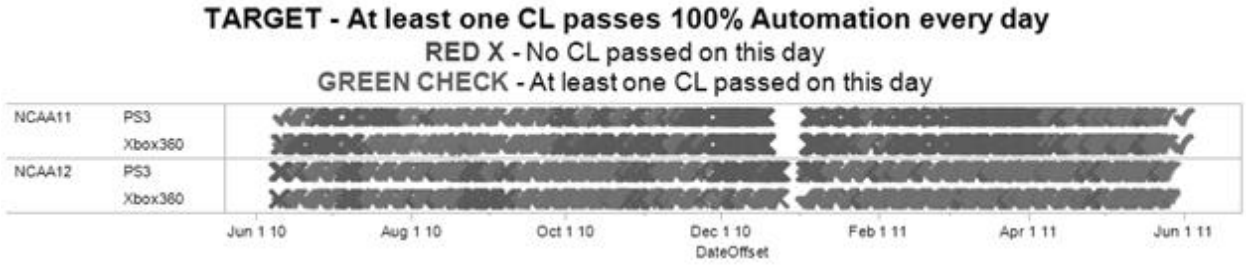


Figure 7 - Smoke Test Stability NCAA 11 & 12

In NCAA11, we went nearly 5 months without getting a passing build. In NCAA12, our worst period was about 3 weeks.

For bots, the picture is similar:

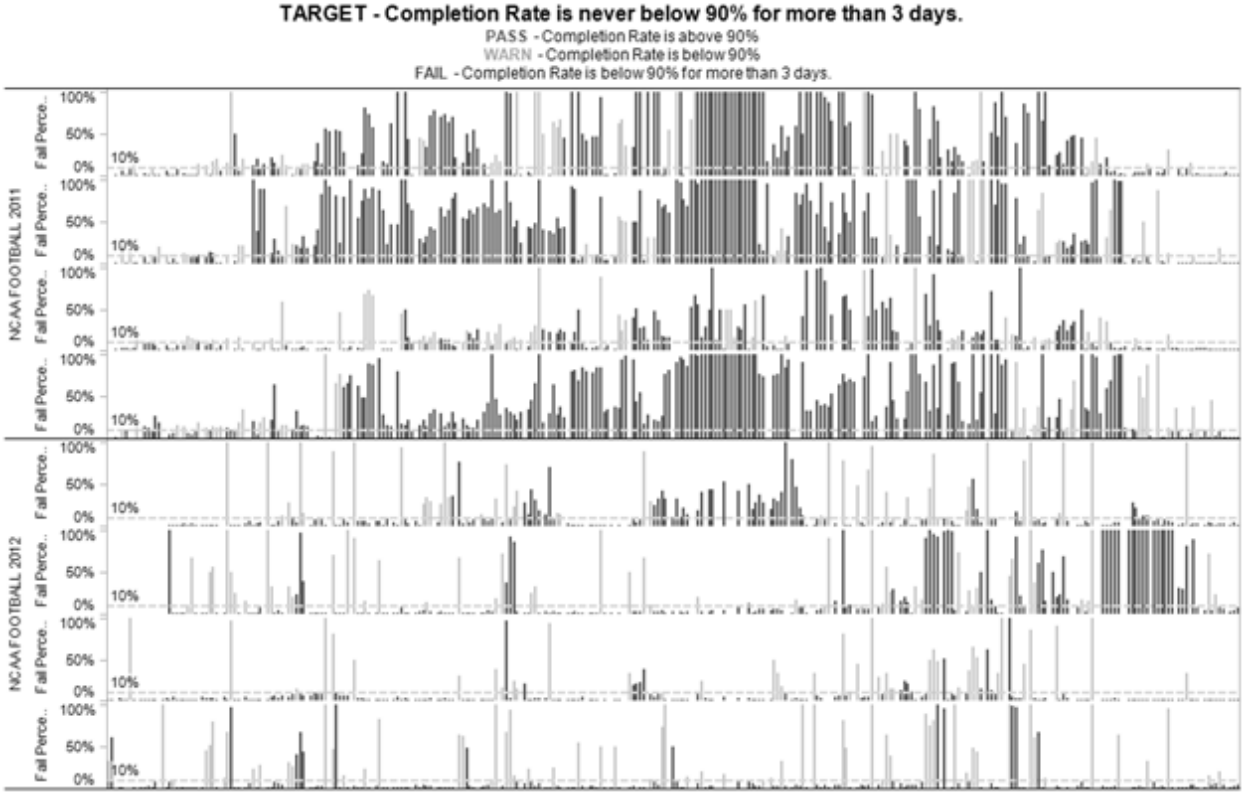


Figure 8 - Bot Test Stability NCAA 11 & 12

In NCAA11, the bots only became stable after we were deep in Alpha. In NCAA12 the bots were stable for most of the cycle. Note that the large red block at the end of NCAA 12 was due to a network outage that was beyond the team's control.

Of course, our real objective was to improve Alpha and the quality of the game. As indicated by this graph comparing Alpha hours between that title and previous iterations, we succeeded here as well. The Finaling team was typically composed of the more senior engineers. As we approached the end of alpha, the Non-Finaling team started to wind down, and all changes in the last few weeks were made by members of the Finaling team.

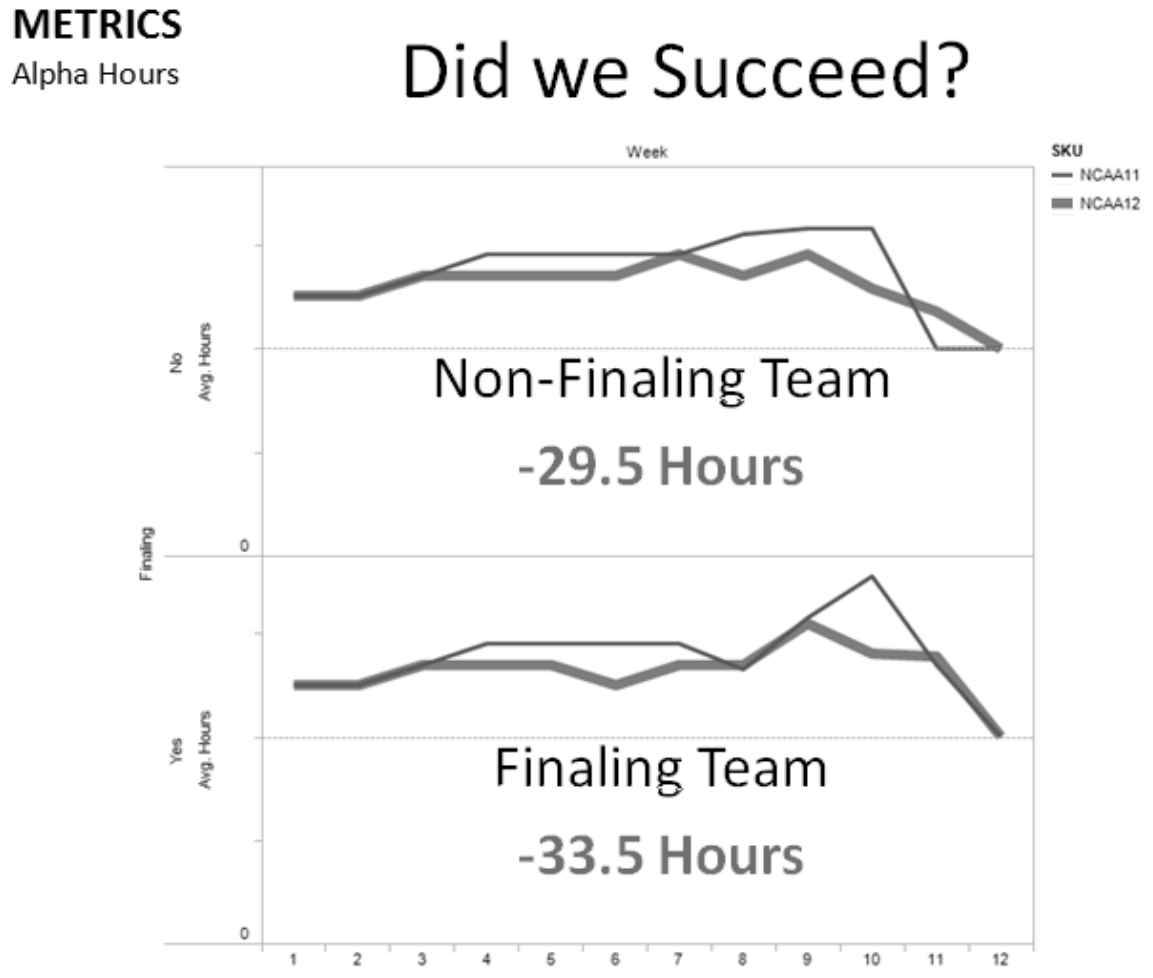


Figure 9 - Alpha Hours NCAA 11 v/s NCAA 12

As you can see from the above graph, both teams ended up working less extra hours during Alpha. This was especially true for the finaling team.

Another way of looking at the results is the graph of Alpha bugs over the entire cycle – again comparing with previous years:

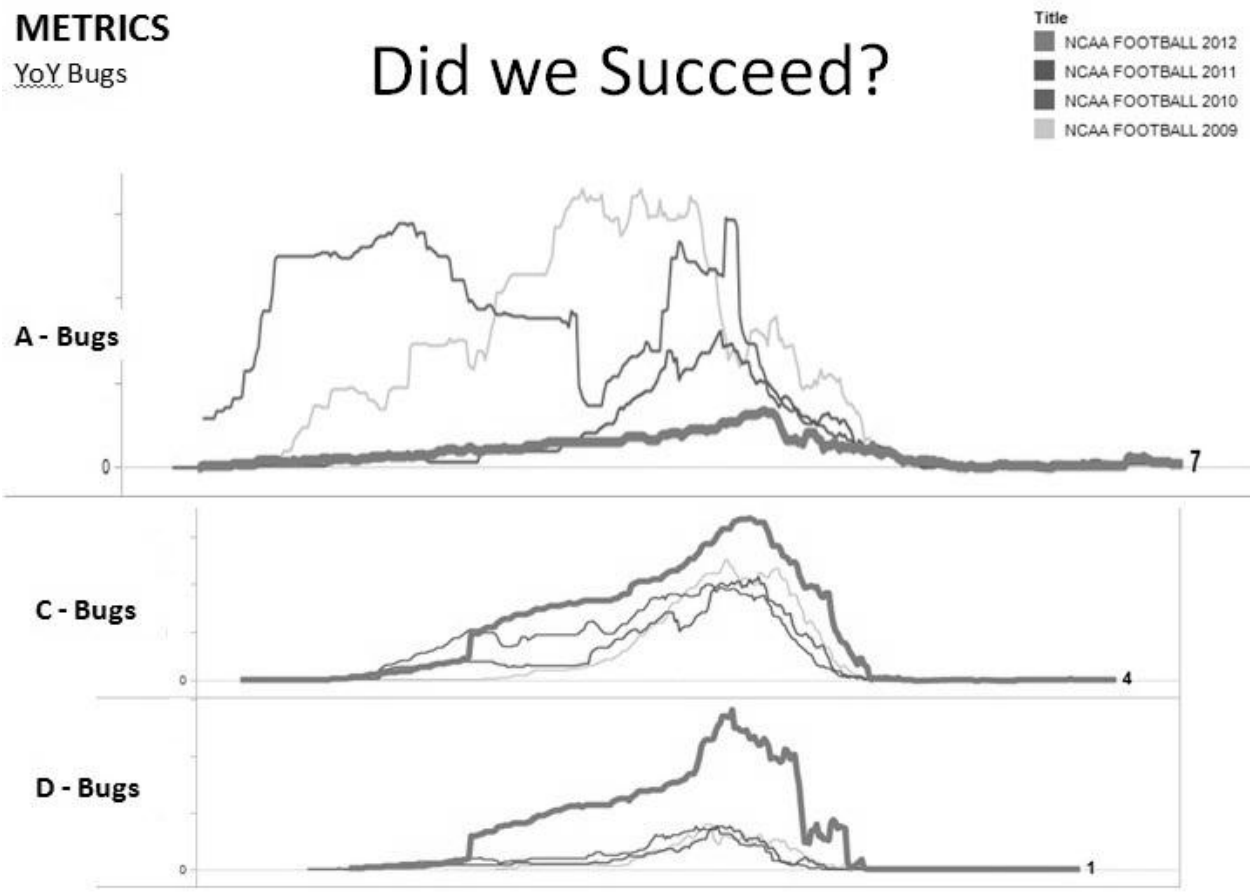


Figure 10 - Year on Year Bugs

Note the drastic reduction in severe (A) bugs from previous years. On the other hand, C and D bugs actually increased (C bugs are defined as medium level – e.g. player names not being sorted in alphabetical order, and D bugs are very minor issues like punctuation). However, A bugs are often blocking bugs that prevent QA from testing areas of the game and finding the lower severity bugs that need to be fixed to make a polished game. By reducing the amount of severe bugs, we ensured that QA had the time and ability to find the lower severity bugs.

We believe that the aspects of the project covered in this paper were all important to achieving the results. If we did not have the infrastructure to support this or the management buy-in or enforced the changes in work habits by taking drastic action (freezing the code-base), this would not have worked.

References

1. Liker, Jeffrey; Hoseus, Michael; Center for Quality People & Organization, 2008. Toyota Culture: The Heart and Soul of the Toyota Way. McGraw-Hill.
2. Rother, Mike, 2009. Toyota Kata: Managing People for Improvement, Adaptiveness and Superior Results. McGraw-Hill.
3. Jones, Capers; Bonsignour, Olivier, 2012. *The Economics of Software Quality*. Addision-Wesley.
4. Boehm, Barry; Basili, Victor R., 2001. *Software Defect Reduction Top 10 List*. IEEE Computer, Volume 34, January 2001, p135-137
5. Huizinga, Dorota; Kolawa, Adam, 2007. *Automated Defect Prevention: Best Practices in Software Management.* John Wiley & Sons.

Deriving Testing Strategies from User Experience Design Practices

Venkat Moncompu
venkatams@yahoo.com

Abstract: User-experience-design (UXD) and Quality-by-Design (QBD) both involve a shared tool-set. Both methodologies use conceptual, behavioral and interaction models to facilitate understanding the user needs and the context of use. This paper suggests that an exploratory tester can exploit the synergies implied by the shared tool-set.

Using a case study, this article attempts to demonstrate how the insights gained through the analysis and review of the UX design artifacts aids the discovery and learning during exploratory testing. It is compelling to see the usefulness of scenario-based contextual design in providing a foundation for unscripted exploratory testing. The case study further shows the gap that exists in these teams rich in engineering culture that is at odds with the exploratory testing which is more of an art than science.

Biography: *Venkat Moncompu is a manager with West Monroe Partners, LLC where he is advocating quality by design in the software development projects that he engages in. Venkat has 15 years of experience in the software industry and interests include quality management, agile test automation, user experience and design thinking. West Monroe Partners, LLC is an international, full-service business and technology consulting firm focused on guiding organizations through projects that fundamentally transform their business.*

Venkat has a master's degree in Engineering from Arizona State University. Venkat has published articles and spoken at various forums on Quality Assurance in Information Technology

Copyright. Venkat Moncompu, 2012.

1. Introduction

Quality-by-design is based on the premise that quality is built-in and verifying quality early in the design reinforces the conformance to quality as the solution is built. UX design methodology starts by reviewing the value proposition and identifying the user segments. The user research then leads to a conceptual understanding of the scenarios and contexts of use in support of the value proposition. The behavioral and interaction design that emerges is a model of the system in the context of its use. The elements that constitute the model are validated by usability tests. These models also drive the user acceptance functional testing when supplemented by task analysis and user goals. Evaluation of the design through usability tests for an understanding of the user interactions are very valuable in discovering new scenarios and gaps in how the system would be used. This is a very powerful and profound recognition of the user acceptance behavior. The findings of the usability tests and insights gained on the user behavior provide guidance to exploratory tests which are a supplement to requirements driven testing.

2. Acceptance Testing

The acceptance testing validates the fitness for use of a system. The factors that influence these usability tests are

- The environments and operating conditions of use,
- The prioritized user scenarios that provide highest value to the user and
- The attributes of quality that satisfy the specific user's needs in the context of their objectives

The motivations, user goals, user roles, functional understandings of the domain and usability tests guide a user centric design process. By iterating through prototypes and high fidelity models, the navigation flows through the system forms the basis of the acceptance tests. User stories build the relationship between user tasks and the goals by mapping user roles and personas to them.

The software engineering team creates elaborate documentation in waterfall methodology that fail to capture these user perspectives at all. We run into limitations when we try to capture these behavioral characteristics in any number of use cases and requirements. However, guided by the UX design these limitations are overcome by actively engaging the testers with the design teams up-front during the evolution of the design (co-location, pair-programming). The requirements in agile development methodology are almost identical to the UX design work items. They are merely presented in a slightly different manner. Therefore, it seems logical to engage the testers with the UX design team earlier in the design process to identify these acceptance conditions, criteria and tests for the definition of what "Done" means.

One of the best practices of agile development is the adoption of acceptance test driven development (ATDD). Unless these are designed as "intended," automated acceptance tests run concurrently as the development team implements the user stories, adds little value to the development effort. The opportunity to design tests efficiently presents itself during the evolution of the UX design itself (lockstep) rather than forced to trail the development as in traditional approaches. The scripted tests rely on the documented requirements which can only specify the systemic view for acceptance. Reviewing the emerging design with the insights gained from user research into the context of their use, the testers bring the acceptance tests to life. The testing team can derive objectives for exploratory tests with a better understanding of the personas, usage scenarios and the interaction model.

During this user research process, additional details of user behavior emerge such as preferences by a certain persona to perform a task and navigation towards a goal which might not otherwise be obvious. The interaction models provide Meta model information that improves the testing efficiency

by uncovering operating conditions, user data needs and motives for navigation flows that are not documented. With unscripted exploratory tests, learning and adapting tests based on the system behavior in the context of its use become intrinsic to the *charter-test-retrospective-adapt* cycle.

Exploratory testing inherently deals with discovering these experiences as the users navigate through the application based on the scenarios and tasks. Given the tangible benefits of front-loading agile development with a robust UX design process, it is ironic that the agile development community has not adopted the practice of UXD in a more embracing manner than they currently do. Part of the problem is that management and the developers focus not on designing creative user experiences but on delivering software engineering service. In the few instances where they do have a UX design practice, there seems to be a dysfunctional relationship between user centric design that is user oriented and solution design that is customer focused where the user and customer are not the same. It then becomes more happenstance than the consequence of a well planned strategy. Secondly, the problem is the inability of the engineering culture of the teams to acknowledge that exploratory testing is as much an art as it is science.

For the benefit of the reader of this article, before visiting the mechanics of exploratory testing to show this correlation in detail and the patterns that emerges therefrom, let me first review the techniques and tools used in UXD in more detail towards deriving acceptance test conditions and criteria.

3. UX Design & Acceptance Testing

The UX designer starts at the very inception of the project by identifying the business value proposition which talks to the uniqueness of the offering and the differentiating attributes of the solution from the competition. The high level objectives of the business problem are evident from the analysis of the value proposition.

These objectives then become the focus areas of higher priority when designing the tasks and test steps around these. The value proposition also discusses the beneficiary i.e. the user base that the solution caters to (emerging market or existing customers). The user segmentations and identification of personas to fit the user segments are then identified during user research. The reasons and motivations of using the system and how the users interact with the system in their working environment are recognized. The pragmatic tester relies on a host of factors such as biases, observations, hunches, instincts, heuristics, and experience to supplement the approach to exploration. To keep these open-ended drivers focused irrespective of the approach (be it free-style, scenario-based, strategy-based or feedback-driven), the understanding of the user and the context of use in delivering value is paramount. For exploration, the tester leverages tools such as checklists, mind-maps, and cheat-sheets to build a framework to base her focus on.

The tester adopts the personas just as the business analyst captures the actors for the use cases. The user tasks lead to the exploration of layout and navigation design with low-fidelity prototypes to begin with even before higher fidelity working prototypes are built. At this juncture, the acceptance tests in concrete system interaction steps begin to emerge. Before fleshing out the details, the tester outlines a matrix of tasks against goals and user personas to form a dimensional matrix of test coverage.

User stories are the preferred tool for capturing requirements because they help describe the users' view of the system behavior rather than a system-centric view described in use cases. Whereas use cases serve as a contract between the various development teams and the stakeholders, user stories are flexible and serve as a negotiable framework for engaging the users to describe user-centric view of the system behavior. User stories are typically takes the form a construct such as:

"As a <role> I want to <goal> so that <benefit>"

What remains then is the acceptance criteria which are the objectives derived from task analysis. It is here that significant contributions have been made in the recent years to design by specification. For ATDD, tools that support Gherkin like domain specific language (DSL) are introduced and the acceptance tests created as shown in example 1.

Given an amateur bird watcher has launched the application
And has browsed to the bird she has spotted in the gallery
When the user begins to record her observations
Then the system should pre-populate all of the information presented to the user in the gallery

Example 1: Using gherkin DSL to capture acceptance test criteria

What is obvious from this example is that the construct is essentially in English language where the user role is clearly defined and the task she is trying to accomplish is also captured. The user research helps fill the role of the user with the appropriate persona for the task's objective. Then the conditions for acceptance of performing this task within the context are identified. The ingredients are essentially the same – a user goal, the task steps and the acceptance conditions in the form of expected behavior. This then becomes specification by test (or example). These acceptance tests are scripted with the steps known *a priori*.

Exploratory testing on the other hand is the unscripted testing of the system by continuous learning and discovery of the system behavior under conditions of use. The factors that differentiate it from *random* testing are the insights testers learn to keep the tester focused on the specific navigation patterns and flows that users take while interacting with the system. The testers rely on heuristics, biases, observations, hunch, intuition, conjectures to comprehend the user's perception of the interface to accomplish their stated goal. The different types of exploratory tests are freestyle, scenario-based, strategy-based and feedback-driven. To provide a structure and framework around exploration, it is very important for the testers and developers to engage with the designers early even as the user experience team begins the user research process. By studying the emerging user personas whose task analysis drive the interaction design, testers derive testing strategies and put the system through its paces in the context of their use. A few different time-boxing techniques have been proposed such as session-based testing where the testing charter is chosen so as to fit into each session of a specific time interval to do the exploration. In the case study that follows, I attempt to derive information based on user research and supporting user experience design artifacts such as wireframes to gain insights into how the system would be acceptable for use.

User stories cross-referenced in this way with the matrix and supporting document such as checklists and mind map help the tester to test scenarios of use by the different user profiles exhaustively. Focusing on the goals, the tester can adapt the tests with the feedback from the system behavior. Tests are then vectored to be more effective but within the realm of possible acceptance of use.

4. Case study

The mobile application under consideration is the bird spotter application that caters to at least 3 different user archetypes. An amateur who is not familiar with bird species, a researcher who knows bird species by name and characteristics, and an avid bird watcher interested in the seasonal migratory patterns and tracks bird migration through social networking channels with others in the spotter community.

A closer look at these personas reveals quite a lot about the individual user needs and the way the application might be used. Based on the personas captured by the UX design team, the environment in which they might be using the application can be inferred.

	**Amateur bird watcher** likes to feed birds for example while out and about with his son. Knows little about birds and therefore has to use the browsing features in the application to help identify birds by their visual and audio characteristics. He would use the application in an urban setting or residential neighborhoods and therefore would only be interested in birds that are commonly seen.
	The **avid bird watcher** wants to spot exotic birds that are seasonal and migratory. He wants to share the spotting information with others in the community. Looking to use the application while on the road or outdoors. He would like to report his observations to the conservatory once he returns home or office and so caching the recorded observations to the device storage would be important.
	The **research student** wants to complete the research on birds and complete the study with spotter data and information. She will be using the application in camp sites and remote areas. She would like to collate all the information she has captured over time and map patterns by geographic location. And so, she would like to navigate the application in such as manner as to derive geographical information to the observations and sightings.

Table 1: Personas for the bird spotter application

In order to understand the application states better, the analysis of the user personas reveals additional requirements: presenting such information as seasonal migratory patterns and a sampling of their bird song and images of the bird to help user in identification.

The avid bird watcher wants to get real-time data feeds of spotting information in her vicinity through a social media channel. She would want to control access to her friends and filter by attributes such as distance or relevance. Here also, we see ample opportunity to explore the application based on the user needs.

The research student persona likes to record detailed information as the lower bound on the spotter count, specific species of interest, direction of migration, geo-tagging of location to compare with a seasonal database such as those maintained by the Audubon Society, for instance.

Based on the layout and interaction design, additional tests can be explored given these additional contexts of use aided by the personas. These additional features increases complexity and the exploratory tester has many opportunities to improve testing efficiency.

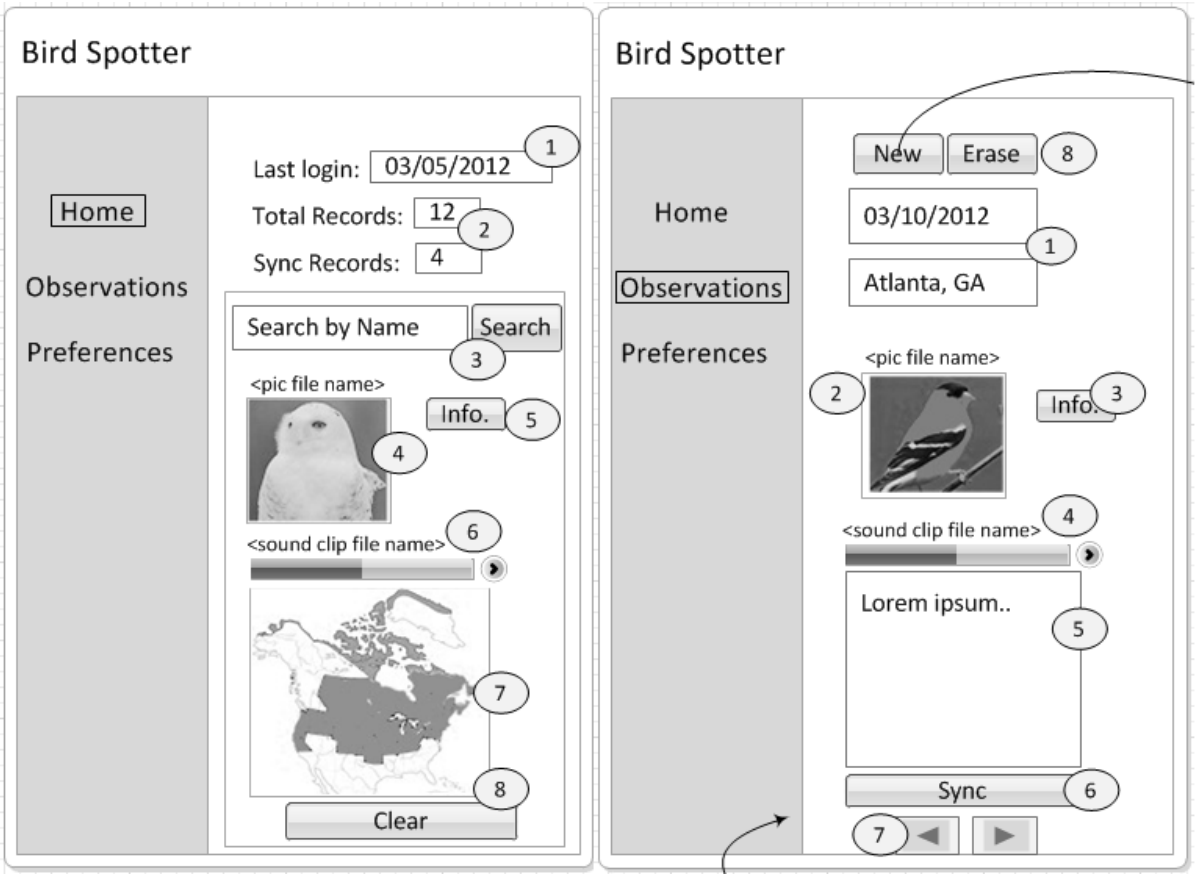


Figure 1: Mock-up of bird spotter application

Given that these users cannot be stationary and tethered, the mobile platform becomes the primary platform of choice. The user interface (UI) and layout design has to factor in such constraints as:

1. The smaller screen has limitations of real-estate for the controls
2. Consider the type of controls from an affordance perspective
3. Evaluate navigation and find-ability based on ease of use – kids, for e.g., would like to use and may not have very adept fingers or retention capability

The UI wireframe for the above user scenarios may look like that shown in figure 1. The UI mock-up is expected to support the user behavior stemming from user research. Combined with the user tasks, these navigations and interactions result in exploratory test flows. As the design iterates to the optimal, the tester can understand the user motivations to drive the testing strategy. The layout and structure of content corresponds to user's task goals and can be validated using acceptance tests. The tester seeks to identify gaps in the tasks specific user roles perform that are missed and tries to relate the user story to the navigation flows that they map to. For instance, a set of administrative tasks to *manage* the amount of content that can be downloaded to the mobile device given its constraints for space was a gap that was identified by exploration.

Another example of the detecting issues from exploration was in the "home" screen. The tester found that the interface did not support the amateur user's need to "browse" through the database of birds

to match the observed features such as color; habitat and bird calls to identify a certain species that was spotted.

As the tester begins to understand the context of use, she uncovers specific navigation flows and system behavior resulting in undocumented application states and test conditions. The user characteristics such as domain knowledge also give insights into the way the application would be used. In the case of the amateur bird spotter she is not expected to know the species by their attributes or zoological name. The amateur needs to be able to look up birds through the browse feature that provides information about the species from the database. So the acceptance criteria might be the expectation of being presented with such collateral information (details on habitat and seasonal migratory patterns) that increases the user's confidence in identification.

The navigation flows around ease of finding this information forms the basis of usability testing but it is not as exhaustive as the opportunity presented to the exploratory tester to vary the test steps. The exploratory tester not only relies on feedback but also on such intangibles as experience and intuition. And so, the tester might want to test "what-if" and edge-scenarios around the flow being discussed here. The tester also relies on deductive reasoning from the system behavior she observes in using the system within the confines of the usability model described by the UX design artifacts.

The more experienced spotter, on the other hand, knows the bird species well enough that she will be searching for them not just by name but also by other characteristics. The application should be able to support these two sets of search-and-browse capability. For these specific and different types of user goals, the tester can explore ways to accomplish the task of locating specific birds in the application by changing the application states that affect the searchable set. These search factors are be affected by varying geographic locale and vicinity of the observer. The tester vectors tests by varying these controls from intuition and system response.

The research student in contrast is merely using the statistical capability for reporting her observations collected from the field. She is looking to get a quick record of the observation without having to browse through or search for the type of species spotted. The focus of this exploratory tester impersonating this persona is around the ease of reporting and being able to "tag" her observations appropriately for statistical compilation later. Though the system behavior forms the basis of exploration, the context and goal of ease-of-use testing would not be recognized without the details surrounding the persona.

Given the different ways of using the application, the UX designers provide guidance for such capabilities as configuring the user preferences. This may be in the form of

A. Specifying a location where the mobile user moves to provide geographically relevant bird collection or
B. Filtering instant messages from other spotters in the network by radial distance and ranking those by list of "friends" in the etc.

Given the concern about privacy and security, user research would also uncover the user's need to be able to grant or deny friends' invitation to share instant messages. The tester then sets out to test these possible different behavior permutations of the system. The ability to test these conditions using scripted tests will not be effective no matter how well the documented requirements are. A combination of structured automated and unscripted exploratory testing approach will likely uncover more potential issues. In addition, the cost of creating documented test assets for increasingly complex systems become prohibitively expensive. Besides, the value these scripted tests provide is little to none when it comes to detecting bugs efficiently. The cost of maintenance of these tests due to changes from big-design-up-front syndrome (BDUF) also increases.

Repeatedly running the same tests to uncover new defects is ineffective just like walking on the footsteps of another in a minefield which is unlikely to set off any explosions. The spirit of exploration

in uncovering new bugs and potential defects is therefore very compelling for reducing inefficiencies in the testing process.

Another gap uncovered by exploratory testing is a scenario when a novice user moved to another geographic location and the application did not trigger update to the list of birds that are common in the new location. The documented requirements only called out the ability of the user to specify the location by using the preferences screen. It is only when the scenarios were reviewed based on tester experience and the application was already in use that the need for prompting the user to change the location was uncovered as an issue.

5. Conclusion

A key component of UX design is user research that results in segmentation of users into user personas. Personas are very useful in the discovery and understanding of the way the system would be used. Quality by design focuses on this aspect of user behavior illustrated by scenarios in the context of their use.

Scenario-based tests that are adaptive to different aspects of system behavior are more effective than any combination of scripted tests planned *a priori*. The personas and user scenarios provide the framework for this form of test adaptation to be contextual and relevant. Providing visibility to the test results from the perspective of user acceptance helps the team evaluate whether the development has achieved the minimally shippable feature set objectively.

For sprint planning and deriving test strategies, the lessons learned from exploration helps not only to develop test charters but also to adapt tests based on feedback. Overlaying the contextual information to assess the impact of defects to specific personas helps to evaluate the quality of the system from a fitness-for-use perspective.

In agile software engineering projects, the testing team can collaborate with the UX design team early in the development phase to converge to the goal of fitness-for-use efficiently. The solution team thus builds-in quality and improves testing efficiency. With pair programming as described above, the testing team also gets to engage early and test often. Adapting tests based on the feedback from system behavior is more realistic when personas are the basis for adaptation. Testing within the context of use in this manner makes the solution more successful.

References

1. Juha Itkonen, Mika V. Mantyla and Casper Lassenius, *Defect Detection Efficiency: Test Case Based vs. Exploratory Testing*, First Intl. Symposium on Empirical software Eng. and Meas. IEEE Comp. Soc., 2007.
2. James Bach, *To repeat tests or not to repeat,* http://www.satisfice.com/blog/archives/24.
3. Adam Goucher, *Quality Through Innovation*, http://adam.goucher.ca/?p=479.
4. James Bach, *Session Based Test Management,* http://www.satisfice.com/sbtm/.
5. Lisa Crispin and Janet Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*, Addison-Wesley Professional, 2009.
6. Jakob Nielsen and Rolf Molich, *Heuristic evaluation of user interfaces*, CHI '90 Proceedings of the SIGCHI conference on Human factors in computing systems: Empowering people, pp. 249 – 256.
7. Jonathon Kohl, *Demystifying Exploratory Testing*, http://www.stickyminds.com/BetterSoftware/magazine.asp?fn=cifea&ac=439.