TWENTY-NINTH ANNUAL PACIFIC NORTHWEST SOFTWARE QUALITY CONFERENCE

PNSQC™

October 10-12, 2011

World Trade Center Portland
Portland, Oregon

PNSQC™ 2011 President's Welcome

Delivering quality does seem to be what it's all about, right? Well, there's also getting people to know about the product; there's motivate them to pay for it; and there's the challenge of staying ahead of the competition. So, maybe delivering quality is not the whole picture, but without quality as a prerequisite, cost, time-to-market and even early sales volume may not matter in the end. Hence our theme this year: 'Delivering Quality'. We think it's one of the biggest objectives to drive useful discussions, analyses, and reports. To address it, we believe a multi-disciplinary gathering of professionals focused on sharing experiences, analyses, learning, and new perspectives is both stimulating and rewarding.

Welcome to the 29th Annual Pacific Northwest Software Quality Conference! In addition to a set of very high quality paper submissions, and still more poster papers, we now also have a website forum to enable continued discussions and interactions begun or raised at the conference.

We are excited to have Goranka Bjedov of Facebook kick off the conference by examining how the growing and changing world of software and customer demands will change the world of quality and testing. Goranka is a bold and insightful thought-leader in software today. Her prior experience includes working for Google, AT&T Labs, and as an Engineering Professor at Purdue. She has also written two textbooks. Based on over two years of research and hundreds of interviews, the insights and suggestions in her keynote will provoke a valuable perspective on what we need to think about to deliver quality in the future.

Our second keynote, Rob Sabourin, has a rich background in software engineering, software management, consulting, and teaching. With over twenty-five years of management experience leading teams of software developers, he is an expert at understanding team factors that lead to success. In his talk, Rob will share stories of successes, failures, and turn-arounds that explain why and how teams need to synchronize on value in order to deliver quality.

Following very positive feedback from last year, we will again have the larger, expert-led, interactive lunch discussions. Our Monday lunch will have five expert-led 'Deep Dive Birds of a Feather' discussions to choose from. The Tuesday lunch will have "Group Therapy", "Ask-An-Expert", "What is QA Like at Your Company?", "Lightning Talks", and an Open Forum on delivering quality. Monday evening's social, which is open to the public, includes both exhibitors and a large number of high quality poster papers. The Tuesday evening Rose City SPIN presentation brings brain science to software productivity. Descriptions of these activities will be broadcast on Twitter. Also, check out our new online forum on www.pnsqc.org. You'll be glad you did!

As a participant at this year's conference, you are strongly encouraged to interact, raise questions, and share insights and experiences. We hope you find the environment stimulating and the experience rewarding.

Bill Gilmore

President, PNSQC™ 2011

TABLE OF CONTENTS

Welcome i

Board Members, Officers and Committee Chairs vii

Additional Volunteers viii

PNSQC Online Community ix

PNSQC Call for Volunteers x

Keynote Address – October 10

The Future of Quality 1
Goranka Bjedov

Keynote Address – October 11

Value Sync 9
Rob Sabourin

Invited Speakers – October 10

21st Century Requirements Engineering: A Pragmatic Guide to Best Practices 21
Erik Simmons

Liftoff: Starting New Projects on a Trajectory Toward Success 41
Diana Larsen

Soft Skills Track – October 10

How to Deal with an Abrasive Boss (aka "Bully") 53
Pam Rechel

Before We Can Test, We Must Talk: Creating a Culture of Learning Within Quality Teams 55
Kristina Sitarski

Delivering Quality One Weekend at a Time: Lessons Learned from Weekend Testing* 61
Michael Larsen

* This paper was accepted by the Program Committee but Michael Larsen is unable to present it at PNSQC™ 2011.

Testing Track – October 10

Delivering Relevant Results for Search Queries with Location Specific Intent **73**
Anand Chakravarty and John Guthrie

An Introduction to Customer Focused Test Design **85**
Alan Page

Testing Services in Production **97**
Keith Stobie

Exploring Cross-Platform Testing Strategies at Microsoft **107**
Jean Hartmann

A Distributed Randomization Approach to Functional Testing **117**
Ilgin Akin and Sam Bedekar

Application Security for QA Managers – Pain or Gain **129**
Dr. Ravi Kiran Raju Yerra

Process Improvement Track – October 10

Reliability Before You Ship **137**
Wayne Roseberry

Creating a Lean, Mean Client Integration Machine **153**
Aaron Akzin, Shelley Blouin, Kenny Tran, Supriya Joshi,
Sudha Sudunagunta and Aaron Medina

Design for Delight Applied to Software Process Improvement **163**
John Ruberto

Releasing Software (How do you know when you are done?) **173**
Doug Whitney

Inspiring, Enabling and Driving Quality Improvement **193**
Jim Sartain

Software Technology Readiness for the Smart Grid **203**
Cristina Tugurlan, Harold Kirkham and David Chassin

Lifecycles Track – October 10

Volunteer Armies Can Deliver Quality Too: Achieving a Successful Result in Open Source, Standards Organizations, and Other Volunteer Projects **213**
Julie Fleischer

Sabotaging Quality: Sacrificing Long-Term for Short-Term Goals ... **221**
Joy Shafer

Increasing Software Quality with Agile Experiences in a Non-Technically-Focused Organization **237**
Aaron B. Hockley

Unusual Testing: Lessons Learned from Being a Casualty Simulation Victim **251**
Nathalie Rooseboom de Vries van Delft

Kanban — What Is It and Why Should I Care? **263**
Landon Reese and Kathy Iberle

Audit Effectiveness — Assuring Customer Satisfaction **275**
Jeff Fiebrich, Diane Clegg and Simon Lang

Invited Speakers — October 11

Standards and Deviations: The Role of Routine in Testing **283**
Michael Bolton

Personal Kanban: Visualizing, Understanding, and Communicating Your Work Load **293**
Jim Benson

Soft Skills Track — October 11

The Ladder of Unmanaged Conflict **295**
Jean Richardson

Learning Software Engineering - Online **303**
Kalman Toth

Hard Lessons About Soft Skills — Understanding the Psyche of the Software Tester **313**
Marlena Compton and Gordon Shippey

Building a Culture around Exceptional Quality **323**
Tracey Beeson

Automation Track — October 11

QUnit JavaScript Testing in the Enterprise **331**
C. David Lee

YES! You CAN Bring Test Automation to Your Company! **351**
Alan Ark

Unit Testing a C++ Database Application with Mock Objects **363**
Ray Lischner

Playback Testing Using Log Files **375**
Vijay Upadya

Green Lantern Automation Framework **381**
Sridevi Pinjala

Parameterized Random Test Data Generation **395**
Bj Rollison

Testing Track — October 11

Testing in Production: Enhancing Development and Test Agility in Sandbox Environment **411**
Xiudong Fei and Sira Rao

Application Monitor: Putting Games into Crowd-sourced Testing ... **429**
Vivek Venkatachalam, Marcelo Nery dos Santos and Harry Emil

No Test Levels Needed in Agile Software Development! **441**
Leo van der Aalst

Application Compatibility Framework — Building Software Synergy **455**
Ashish Khandelwal, Shishira Rao and Amrita Desai

Process Improvement Track — October 11

Watch your STEP! **469**
Prabu Chelladurai

Reverse Engineering: Vulnerabilities and Solutions **487**
Barbara Frederiksen-Cross and Susan Courtney

Dirty Tricks in the Name of Quality **501**
Ian Dees

Performance Track — October 11

Delivering Quality in Software Performance and Scalablity Testing **513**
Khun Ban, Robert Scott, Kingsum Chow, and Huijun Yan

Perf Cells: A Case Study in Achieving Clean Performance Test Results **521**
Vivek Venkatachalam, Shirley Tan and Marcelo Nery dos Santos

PNSQC™ BOARD MEMBERS, OFFICERS, and COMMITTEE CHAIRS

Bill Gilmore – Board Member & President

Brian Gaudreau – Board Member & Vice President
Delta Dental

Bhushan Gupta – Board Member & Secretary
Nike, Inc.

Doug Reynolds – Board Member & Treasurer
Tektronix, Inc.

Paul Dittman – Board Member & Program Co-Chair

Les Grove – Board Member & Program Co-Chair
Web MD

Sion Heaney – Board Member & Community Outreach

Jon Bach – Invited Speaker Chair
eBay

Bill Baker – Program Co-Chair

Sue Bartlett – Marketing Co-Chair

Shauna Gonzales – Conference Format Chair
Nike, Inc.

Dev Lavu – Volunteer Chair
Nike, Inc.

Rhea Stadick – Operations & Infrastructure Chair
Intel Corporation

Kal Toth – Forum Chair

PNSQC™ ADDITIONAL VOLUNTEERS

Rick Anderson

Chris Blain

Darrel Bonner

John Burley

David Butt

Robert Cohn

Susan Courtney

Ian Dees

Moss Drake

Rohit Dudani

Cynthia Gens

Daniel Grover

Leesa Hicks

Jacob Hills

Doug Hoffman

Michael Larsen

Dave Liesse

Roslyn Lindquist

Launi Mead

Jonathan Morris

Bill Opsal

Dave Patterson

Marilyne Pelerine

Ganesh Prabhala

Ian Savage

Jeanette Schadler

Wolfgang Strigel

Patt Thomasson

Mano Vela

Richard Vireday

PNSQC™ Online Community

PNSQC is now offering an online community! Our community **forum** is up and actively being used. With the addition of this forum, PNSQC is more than an annual conference. We are also a year round online community. The PNSQC forum provides a place for you to contact and interact with other software quality professionals through discussion groups, private messaging, and buddy lists. Feedback from past conference surveys showed that attendees wanted more opportunities to network. Now you have it!

Through use of the forum, you can:

- Raise questions or issues that are work-related, technical or professional in some regard by confining the audience to people who are respectful and are serious about software quality;
- Continue discussions about conference content after the conference and throughout the year with speakers including keynotes and invited speakers. You can ask questions of these speakers and perhaps have other interested members join in the discussion.
- Develop peer professional relationships. You are able to maintain a profile and decide what information about yourself you are willing to share.

Forum postings and all other PNSQC website intellectual content (e.g., conference proceedings and abstracts from current and past conferences) is openly viewable to the public. However, only community members (registration is free) can post to the forum and have other interested members join the discussion. And only members can use the private messaging and buddy lists. Though some people have hesitancy about memberships in general, this was set up to enable a channel of more private discussions for a community. Visit the **PNSQC Community** link on our website to learn more.

Building community is a creative and interactive process that allows for a greater whole from the individual parts. We invite you to be part of this creative process. Send us your ideas on what would make a *great* PNSQC community. Please use the **Contact Us** link on our website, select PNSQC Community as the recipient and let us hear your ideas. Announcements about the progress of the community-building process and services will be included in the PNSQC newsletters sent throughout the year.

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 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 online forum. Manages the social networking channels of communication. Recruits and works with incoming volunteers to place them in committees.

The Future of Quality

Goranka Bjedov, Facebook

We have entered an era of infinite complexity – it is impossible to understand even small subsections of software we are relying on, let alone comprehend the complete solutions. What do all of the prevalent trends in software development (rapid, agile, cloud based) mean for quality?

At the same time, customer expectations have aligned with what is available, and they seem to be more interested in the availability of new features and price than quality.

This presentation addresses the impact of these changes on the testing professionals and tries to assess where the future will take us. It summarizes two years of research on this topic, discussions with hundreds of software testing professionals, and provides suggestions on possible solutions.

Goranka Bjedov works as a Capacity Engineer at Facebook. Her main interests include performance, capacity and reliability analysis, testing and planning. Prior to joining Facebook, she spent five years doing performance testing for Google. Her career also includes senior engineer and manager positions at Network Appliance and AT&T Labs, respectively. Before joining the industry, she was an Associate Professor in the Purdue University School of Engineering. A speaker at numerous testing and performance conferences, Goranka has authored many papers and presentations and two textbooks.

facebook

The Future of Quality

Goranka Bjedov
Presented October 10th 2011

Agenda

1. Introduction
2. The Past
3. Testing for Quality
4. Testing for Productivity
5. The Future

My Background

Facebook – Capacity engineer focusing on capacity analysis and performance /monitoring (Linux + OS based)

Google – Performance engineer focusing on exploratory performance testing and benchmarking (Linux based)

Network Appliance – Reliability/performance engineer for Data OnTap, embedded OS (Linux based)

AT&T Labs – Performance Test Manager for "next generation IP platform" (Linux based)

Purdue University – Assistant/Associate Professor

History of This Presentation

2008 – General discontent with work/field/practice…

Sep 2009 – Moved to London (Hoping for a change)

Nov 2009 – Realization: Quality is Dead

Dec 2009 – Invite from Ross Collard to present at "Leadership Summit" at StarEast and StarWest 2010

2010 – Five months of time available for research

Aug 2010 – start at Facebook in a different role

2011 – a year later: some things are more clear…

Agenda

1. Introduction
2. The Past
3. Testing for Quality
4. Testing for Productivity
5. The Future

Definitions

Testing – What is testing? Is it different from checking (Michael Bolton)

Quality – I like Jerry Weinberg's definition: "It is a value to somebody."

What is the responsibility of a test person? – This is, in my opinion, where we really went wrong.

Other Engineering Fields – Are there similarities? Differences?

What happens when:
Value of quality << Price of quality

The Past

Packaged Software – testing/bug fixing cycle follows development which was preceded by design.

Cost of a Bug – bugs found and fixed in design are cheaper then those found and fixed in development, which in turn are cheaper then those fixed in the field.

Test Team Contribution – finding bugs.

Systems – relatively simple.

Code – relatively simple and traceable.

Development/Testing Tools – very primitive.

Testing Roles

Advocate for the Customer – it is our job to make sure nothing escapes.

Quality Consultant – it is our job to provide information on the quality of the software.

Quality Control – it is our job to stop bad products from being released.

Agenda

1. Introduction
2. The Past
3. Testing for Quality
4. Testing for Productivity
5. The Future

Testing For Quality

To understand product quality, you must test the product with that purpose in mind.

Problems:
- Medium and large tests
- Hard to design and write well
- Difficult to debug
- Tests are more flaky as the system complexity grows
- Take long time to run
- Require (smart) people for analysis [expensive]
- Find real and superficial bugs?

Result: We may get a high-quality product…sometimes.
Result: We get an expensive product…almost always.

When is Quality Important?

Easy to state:
- When a field/industry is highly regulated (maybe drug manufacturing?)
- When human lives are at stake (hospital equipment, planes, cars?)
- When security is at stake
- When money is exchanging hands (isn't that what SOX compliance is all about?)
- Take long time to run

Let's take a look at some of these…

Value of Quality

Glaxo case:
http://www.boston.com/business/healthcare/articles/2010/10/27/glaxosmithkline_to_pay_750m_fine_in_fraud_case/

…GlaxoSmithKline PLC agreed to pay $750 million to settle civil and criminal charges…

The settlement, one of the largest ever in a health care fraud case, burnished the reputation of the US attorney's office in Boston as the premier federal office for investigating health care fraud. It has been responsible for recovering about $6 billion in health care fines and claims in the past decade, about 25 percent of all recoveries nationally.

Value of Quality, cont.

Report on accident on 27 November 2008:
http://www.bea.aero/docspa/2008/d-la081127.en/pdf/d-la081127.en.pdf

Do A330 aircraft have issues:
http://www.spiegel.de/international/world/0,1518,766148,00.html

Not just aircraft:
http://blogs.crikey.com.au/planetalking/2010/11/17/the-anatomy-of-the-airbus-a380-qf32-near-disaster/

Quote: Rolls-Royce had designed and was introducing a fix for the oil leak issues for this into the engines at its own speed. Qantas was left in the dark.

Value of Quality, cont.

It goes on and on:

- Amazon data center down April 21, 2011
  http://www.bbc.co.uk/news/technology-13160929
- Gmail deletes user data for 150K users
  http://www.crunchgear.com/2011/02/28/storm-clouds-gmail-failure-reinforces-danger-of-becoming-too-cloud-dependent/
- Flikr deletes wrong paid accounts
  http://techcrunch.com/2011/02/02/flickr-accidentally-wipes-out-account-five-years-and-4000-photos-down-the-drain/
- Medical devices (personal experience)
- Nodar Kumaritashvili, Feb 12 2010

Some Entertainment

Cable bill - $16.4 million:
http://www.daytondailynews.com/business/time-warner-charges-wright-patt-engineer-16-4-million-for-cable-1117224.html

While the dollar amount of DeVirgilio's billing ordeal makes it among the more egregious in recent memory, the kings of all billing mishaps have to be... Jon Seale, received notice that he owed a 17-figure sum that totaled almost 2,000 times the national debt: 23 quadrillion, 148 trillion, 855 billion, 308 million, 184 thousand and 500 dollars. The other, Josh Muszynski, was charged 23 quadrillion after buying a pack of cigarettes at a gas station.

Agenda

1 Introduction
2 The Past
3 Testing for Quality
4 Testing for Productivity
5 The Future

Productivity Testing

My Definition: Productivity testing is all testing that is focused on faster development. Its main purpose is preventing developers from checking in bad code, and allowing for changes to the code with certain "peace of mind".

Typical examples: Unit tests, micro-benchmarks, "sanity" tests, etc.

Typical characteristics: small, fast, cheap to write and maintain, analysis-free (when they fail, it is clear where and why), perfect for automation, fantastic for gaming the system, managers love them (code coverage, metrics), "technical" (are we using mocks, fakes, doubles, or something else?)

Productivity Testing, cont.

Extremely Popular:
- search for testing framework: ~5 million results (March 2010)
- offers: for Java, for C, for C++ in search box
- language based
- automated testing conference: ~2 million results

Advantages:
- quick to write
- easy to maintain
- fantastic for generating metrics

But, what do they really do?

Testing By Developers Exposed

http://www.exampler.com/ease-and-joy.html

Quote: "The lore of testing is full of people who spent weeks improving test automation libraries without ever, you know, quite getting around to automating any tests. The trick is to make improvements in small steps while simultaneously continuing to frequently deliver the business value that makes the project worth funding. There's a real skill to moving gradually and continuously and simultaneously toward several larger goals."

It is Usually Done Badly

Check any talks/papers/references. People talk about:
- Number of tests (Relevance?)
- Coverage
- Execution time (Fast = Good)
- Automated generation of tests/data
- Testing tools/harnesses/frameworks etc.

Interestingly, I cannot find any meaningful information on:
- What do your tests do?
- Who analyzes them?
- Do they save or waste money?

Agenda

1. Introduction
2. The Past
3. Testing for Quality
4. Testing for Productivity
5. The Future

To Summarize

1. We want it free, we want it now. We can compromise on how good it needs to be.
2. Cloud services are allowing for "instant" fixes. The cost of fix once software is deployed is NOT the order of magnitude higher than before.
3. Paradoxically, this makes fixing bugs only when encountered (by customers/developers) a reasonable strategy
4. This is happening in many aspects of our lives, not just software.
5. System complexity allows for plausible deniability.
6. As the demand for quality products drops, the price goes up steeply.

Pool of Test People has Expanded

uTest and the like – crowdsourced testing

Security bugs – https://www.facebook.com/whitehat/bounty/ http://googleonlinesecurity.blogspot.com/2010/11/rewarding-web-application-security.html

Hardware testing – search for Google Cr-48 and free

Find a bug – get a job? Impossible?

Not Just People – automated systems http://cacm.acm.org/magazines/2010/2/69354-a-few-billion-lines-of-code-later/fulltext

Where We Can Offer Value

1. We can reduce development time (and costs) by writing productivity tests
2. We can bring in quality by adding smart system tests in the right places
3. Think performance, scalability
4. Think usability
5. We can reduce number of machines needed in data centers
6. We must start calculating and communicating the value of our work (dollar amounts)
7. We must stop being the cost center

Value Sync

Rob Sabourin, AmiBug.Com, Inc.

Can quality products be delivered when teams, customers, users and stakeholders have conflicting values? Rob Sabourin suggests that the notions of “on time, on quality and on budget” are meaningless concepts unless you are “on purpose”.

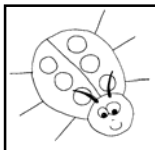
What do people value? Why do they value it? How does it matter?

Rob shares some rich and varied experiences leading successful development projects with synchronized core values between stakeholders, team members, customers and user communities throughout their relationship.

Rob examines some projects that were dismal failures due to teams working at cross-purposes with conflicting values. He also looks at some of the most absurdly turbulent, chaotic, projects that were tamed and became glowing successes due to a deliberate focus on harmonizing a blend of business, technical, organization, team, individual and cultural values.

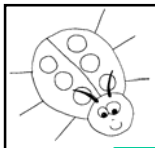
Explore how value sync can make a difference in your context.

Rob Sabourin, P.Eng., has more than twenty-five years of management experience leading teams of software development professionals. A well-respected member of the software engineering community, Rob has managed, trained, mentored, and coached hundreds of top professionals in the field. He frequently speaks at conferences and writes on software engineering, SQA, testing, management, and internationalization. Rob wrote “I am a Bug!”, the popular software testing children’s book; works as an adjunct professor of software engineering at McGill University; and serves as the principle consultant (and president/janitor) of AmiBug.Com, Inc. Contact Rob at rsabourin@amibug.com.



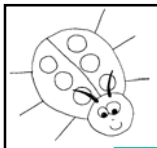
Value Sync

Robert Sabourin
President
AmiBug.Com, Inc.
Montreal, Canada
rsabourin@amibug.com



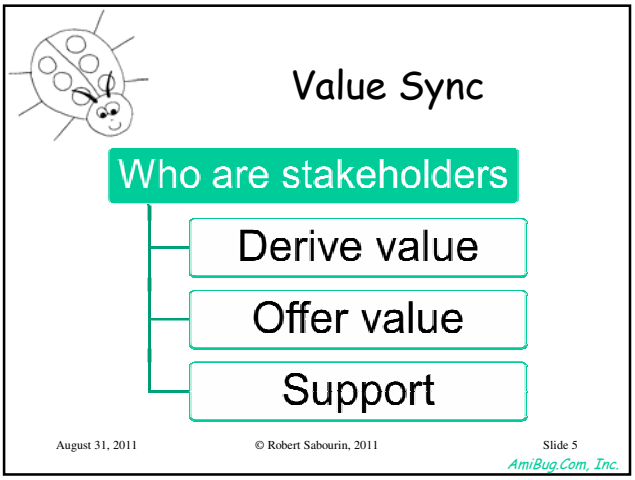
Value Sync

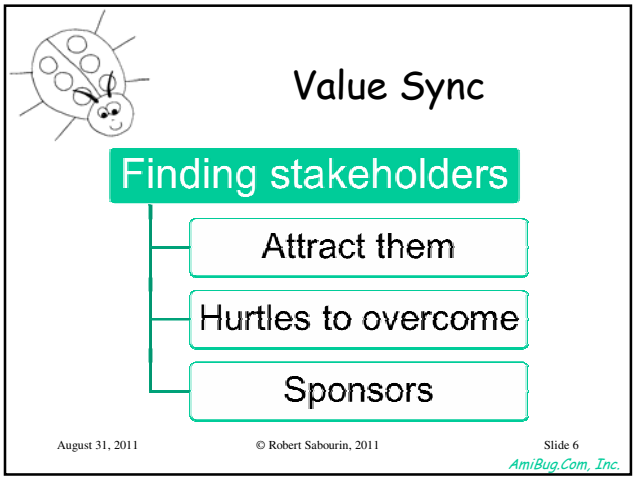
- What is Quality
  - Conformance
  - Suitability
  - Value

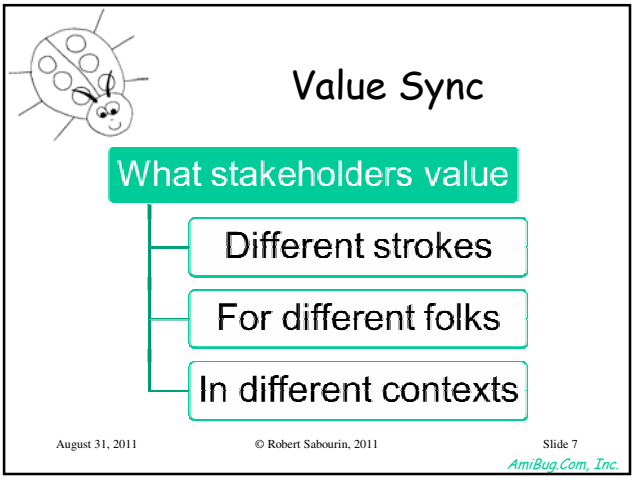


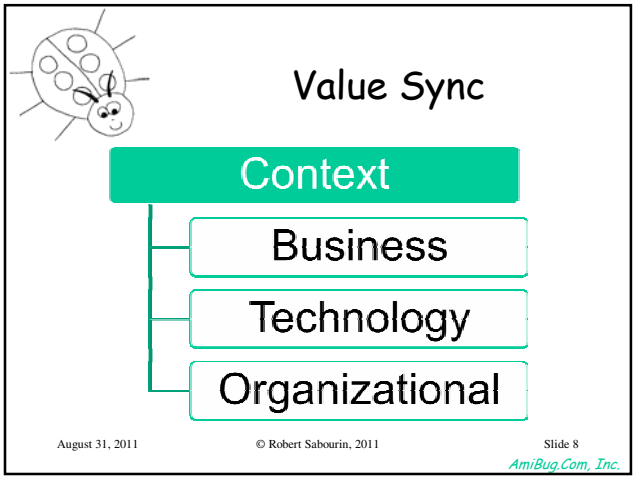
Value Sync

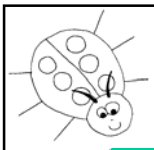
- What is Value
  - Physical
  - Emotional
  - Ethical


Value Sync
Who are stakeholders
Derive value
Offer value
Support
August 31, 2011
© Robert Sabourin, 2011
Slide 5
AmiBug.Com, Inc.


Value Sync
Finding stakeholders
Attract them
Hurtles to overcome
Sponsors
August 31, 2011
© Robert Sabourin, 2011
Slide 6
AmiBug.Com, Inc.

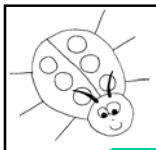

Value Sync
What stakeholders value
Different strokes
For different folks
In different contexts
August 31, 2011
© Robert Sabourin, 2011
Slide 7
AmiBug.Com, Inc.


Value Sync
Context
Business
Technology
Organizational
August 31, 2011
© Robert Sabourin, 2011
Slide 8
AmiBug.Com, Inc.



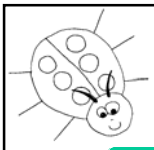
Value Sync

- **Context**
  - Static
  - Dynamic
  - Revealed



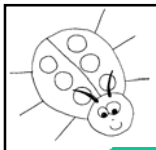
Value Sync

- **Context is alive**
  - Identify sources
  - Actively listen
  - Adapt



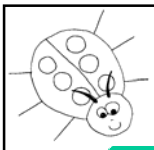
Value Sync

- **Context testing**
  - How does software react to changing contexts?
  - How does context react to changing software?
  - How should testing adapt?



Value Sync

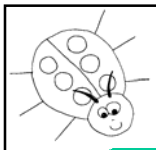
- **Reuse out of context**
  - Good thing?
  - Inherit context dependent decisions
  - We grab code – what of the context?



Value Sync

The right time & place

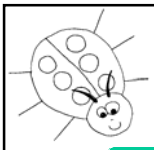
- Some stakeholders fade in and out at different times
- Help them stay on their path
- Use "non intrusive" mechanisms to make sure we hear them and are heard by them



Value Sync

Stuff stakeholders value

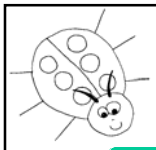
- Material stuff
- Emotional stuff
- Community stuff



Value Sync

Stuff stakeholders value

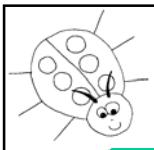
- Ethical behaviour
- Make and keep commitments
- Consistency



Value Sync

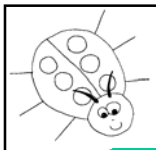
Why test?

- Gatekeepers?
- Information providers?
- Value custodians?



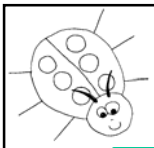
Value Sync

- Help stakeholders see
  - Make visible
  - Stakeholder language
  - Headlights



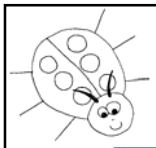
Value Sync

- Challenging Stakeholders
  - "Let the wookie win"
  - Buy in to decision making
  - Derive value from conflicts to drive improvements



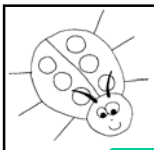
Value Sync

- Value sync
  - Common understanding
  - Shared belief
  - On purpose



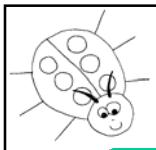
Value Sync

- Purposeful testing
  - On Time
  - On Quality
  - On Budget



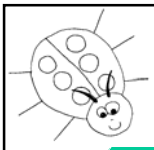
Value Sync

- Do you value objectivity
  - Mission of testing
  - Provide massive objective
  - Information about product and project



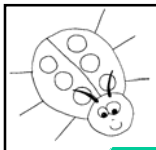
Value Sync

- Value in subjective assessment
  - Your opinion
  - What would critics say
  - What about customers?



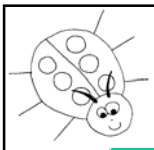
Value Sync

- Learning what stakeholders value
  - Ask
  - Listen
  - Observe



Value Sync

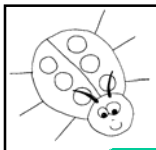
- What testers are good at
  - Measure
  - Assess
  - Learn



Value Sync

Value sync is about

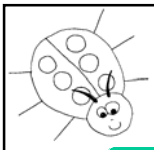
- Knowing what matters
- Understanding why
- Communicating about it



Value Sync

Alternative to value sync

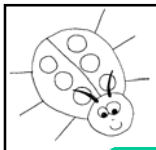
- Walk away
- Find common grounds
- Build on shared values



Value Sync

Economic focus

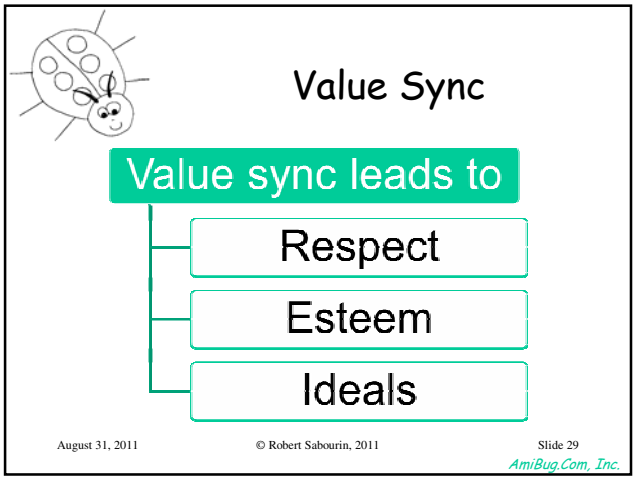
- Effort done and to go
- What we could ship now
- Report downstream costs

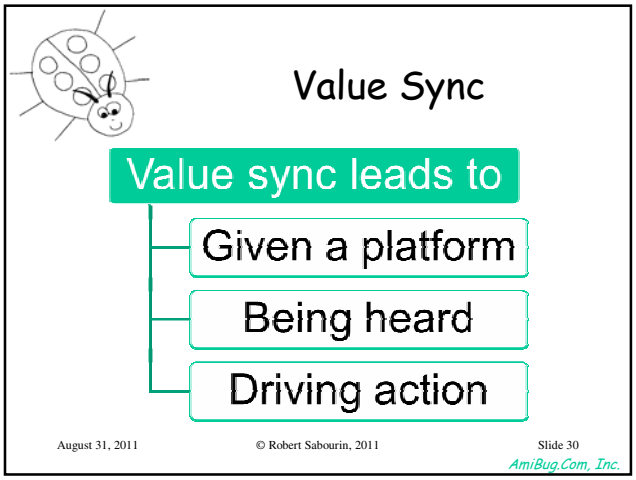


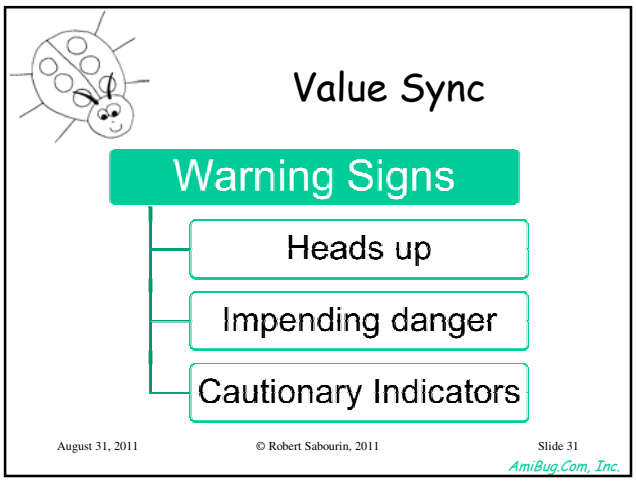
Value Sync

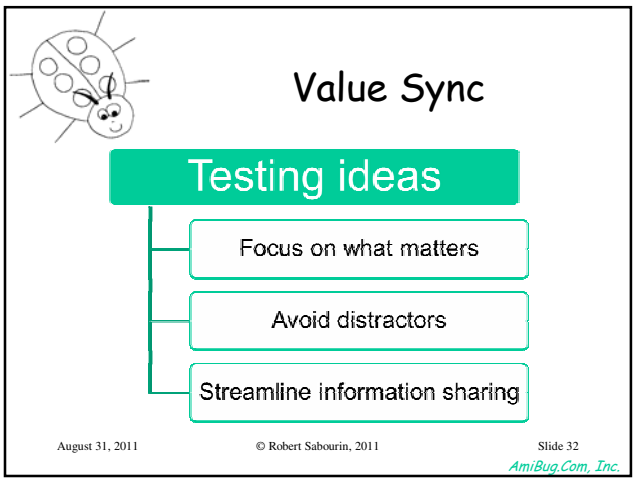
Time focus

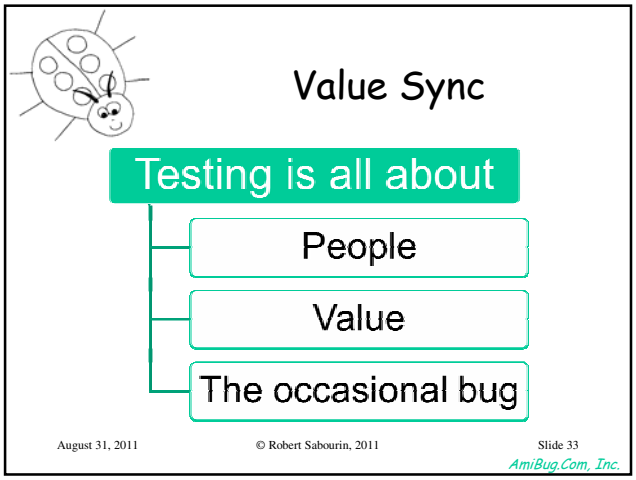
- Trade off workflow
- Manage testing debt
- Maintainability focus

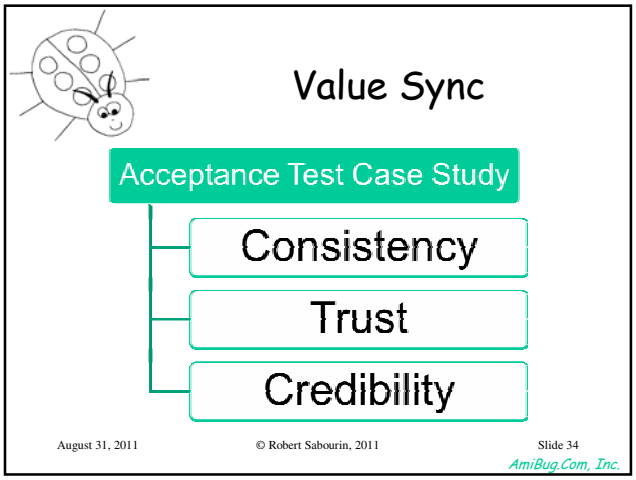

Value Sync
Value sync leads to
Respect
Esteem
Ideals
August 31, 2011
© Robert Sabourin, 2011
Slide 29
AmiBug.Com, Inc.

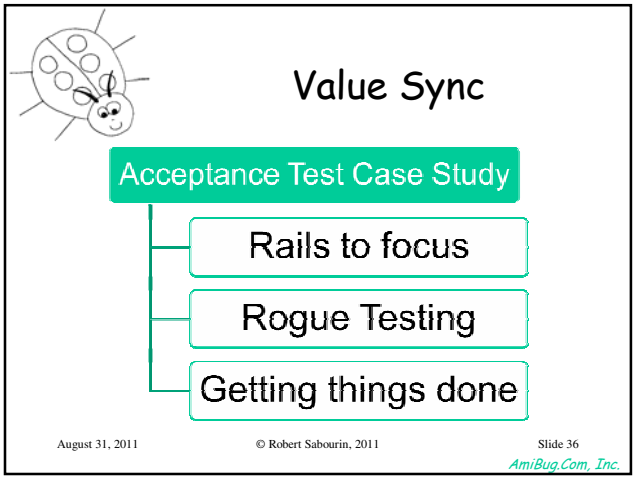

Value Sync
Value sync leads to
Given a platform
Being heard
Driving action
August 31, 2011
© Robert Sabourin, 2011
Slide 30
AmiBug.Com, Inc.

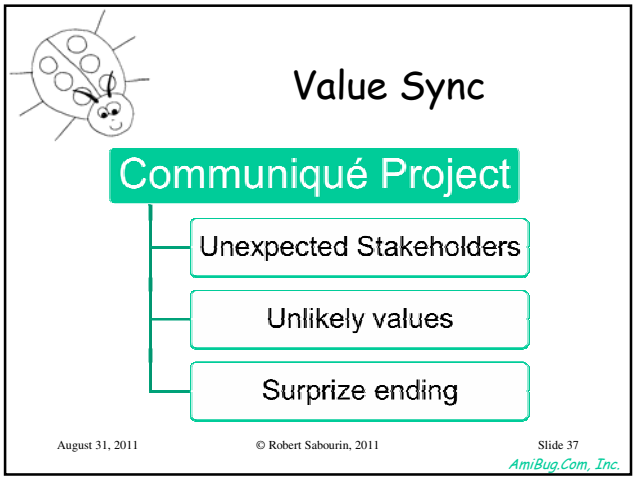

Value Sync
Warning Signs
Heads up
Impending danger
Cautionary Indicators
August 31, 2011
© Robert Sabourin, 2011
Slide 31
AmiBug.Com, Inc.

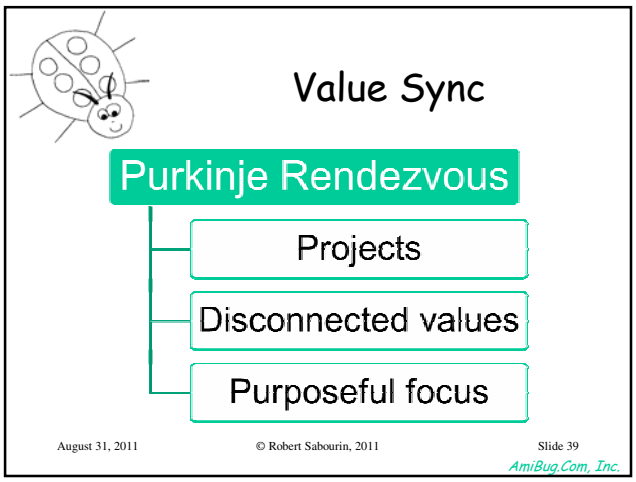

Value Sync
Testing ideas
Focus on what matters
Avoid distractors
Streamline information sharing
August 31, 2011
© Robert Sabourin, 2011
Slide 32
AmiBug.Com, Inc.


Value Sync
Testing is all about
People
Value
The occasional bug
August 31, 2011
© Robert Sabourin, 2011
Slide 33
AmiBug.Com, Inc.


Value Sync
Acceptance Test Case Study
Consistency
Trust
Credibility
August 31, 2011
© Robert Sabourin, 2011
Slide 34
AmiBug.Com, Inc.


Value Sync
Acceptance Test Case Study
Rails to focus
Rogue Testing
Getting things done
August 31, 2011
© Robert Sabourin, 2011
Slide 36
AmiBug.Com, Inc.


Value Sync
Communiqué Project
Unexpected Stakeholders
Unlikely values
Surprize ending
August 31, 2011
© Robert Sabourin, 2011
Slide 37
AmiBug.Com, Inc.


Value Sync
Purkinje Rendezvous
Projects
Disconnected values
Purposeful focus
August 31, 2011
© Robert Sabourin, 2011
Slide 39
AmiBug.Com, Inc.

21st Century Requirements Engineering: A Pragmatic Guide to Best Practices

Erik Simmons, Intel Corporation

Requirements engineering is a core discipline to product development, whether an organization is large or small; involved in market-driven products, IT development, or contractual work; or using traditional or agile methods. There is no shortage of books, papers and courses on requirements, but what really works, and where to start?

In this session, we'll examine some of the core questions that govern how much detail is enough, which areas need it, and when to provide it – regardless of what software life cycle you are using. In addition, we will cover some of the practices that have proven most useful across projects of all types.

So, if you are confused about "agile requirements", can't find the right balance of detail level vs. cost and deadlines in your requirements work, or just want to see some broadly useful practices that you can start using immediately, stop by for the discussion.

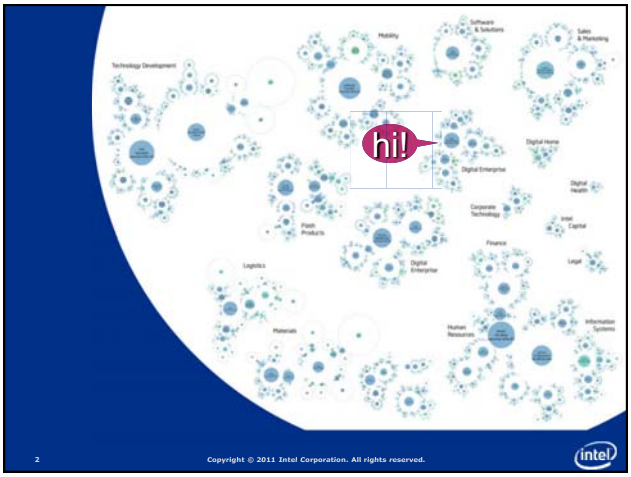
Erik Simmons works in the Corporate Platform Office at Intel Corporation, where he is responsible for creation, implementation, and evolution of requirements engineering practices and supports other corporate platform and product initiatives. Erik's professional interests include software development, decision making, heuristics, development life cycles, systems engineering, risk, and requirements engineering. He has made invited conference appearances in New Zealand, Australia, England, Belgium, France, Germany, Switzerland, Finland, Canada, and the US. Erik holds a Masters degree in mathematical modeling and a Bachelors degree in applied mathematics from Humboldt State University, and was appointed to the Clinical Faculty of Oregon Health Sciences University in 1991.

21st-Century Requirements Engineering: A Pragmatic Guide to Best Practices

Erik Simmons

PNSQC 2011

Version 1.0, 08/11



Contents

Introduction
- Fundamental Concepts
- The Setting and the Challenges

Detail Level and Timing Issues

Requirements Practices for Today's Environment
- The Three-Circle Model
- Specification Basics
- The Easy Approach to Requirements Syntax (EARS)
- Planguage
- The Landing Zone
- Specification Quality Control (SQC)

Sources for More Information

Note: Third-party brands and names are the property of their respective owners

Introduction

What is a Requirement?

A **requirement** is a statement of:

1. What a system must do (a system function)
2. How well the system must do what it does (a system quality or performance level)
3. A known resource or design limitation (a constraint or budget)

More generally,

A requirement is *anything that drives a design choice*

The Purpose of Requirements

Requirements help establish a clear, common, and coherent understanding of what the system must accomplish

Clear: All statements are unambiguous, complete, and concise	**Common**: All stakeholders share the same understanding	**Coherent**: All statements are consistent and form a logical whole

Requirements are the foundation on which systems are built

Well written requirements drive product design, construction, validation, documentation, support, and other activities

Requirements Engineering

Requirements Engineering is the systematic and repeatable use of techniques for discovering, documenting, and maintaining a set of requirements for a system or service.

Requirements Engineering Activities

Elicitation	Analysis & Validation	Specification	Verification	Management
Gathering requirements from stakeholders	Assessing, negotiating, and ensuring correctness of requirements	Creating the written requirements specification	Assessing requirements for quality	Maintaining the integrity and accuracy of the requirements

Current Challenges | Complexity and Pace

Problem Complexity	We've solved most of the simple problems
Solution Complexity	We're not finding many simple solutions to complex problems
Design Tool Complexity	Multi-core, multi-threaded, distributed, cross-platform…yikes
Organizational Complexity	Larger, distributed software teams, more cross-domain interactions and dependencies
Software Development Process Complexity	This is a natural response to increasing solution complexity
Market Forces	The expectations placed on teams have not relaxed, even in the face of the other factors

Current Challenges | Choice and Change

Today's markets are more fluid than ever, and consumers are more willing than ever to shift their thinking, spending, and brand loyalties

Companies must now innovate continuously, or risk loss of customer base to a more innovative rival

- Traditional barriers between product types are falling and new markets are emerging
- Usage models are evolving
- Shorter cycle times mean more threats to market-leading products

For example, think about how many ways music and video can be consumed today

Focus on customer delight and the rapid delivery of value to end users

The Need for Agility

Does Requirements Engineering Matter in an Agile World?

Yes! Complexity and pace mean we have define problem and solution, avoid rework, and maximize reuse

But this can't be "your grandfather's requirements engineering" – 21st-century requirements engineering must be different:

Less	More
Front-loaded, static, stand-alone	Incremental, fluid, integrated
Dictatorial	Collaborative, supportive
Exhaustive, speculative	Just-enough, just-in-time

The question is: How much requirements engineering, and when?

The Need for Abstraction & Hierarchy

Complexity requires better ways to address various views and subsets of a problem or solution

Aspect-oriented development and cross-cutting concerns are good examples

The biggest value in today's systems comes from emergent behaviors, and is not found in any single component

Requirements engineering, done correctly in partnership with architecture and design, can provide helpful abstraction and hierarchy

- What is it that is most valuable in your systems?
- Is that value found in a single component?
- Is it delivered by a single team?

Detail Level and Timing Issues

How Much Detail is Enough?

The correct detail level, like the correct investment in requirements activities overall, must balance risk and investment

The acceptable region of risk and investment differs by product type and many other factors

How Much Detail is Enough?

The correct level of detail in requirements depends on factors that include:

- **Precedented vs. unprecedented product**
- **Development team experience, size, and distribution**
- **Acceptable risk level during development**
- **Domain, organizational, and technical complexity**
- **Need for regulatory compliance**
- **Current location in the development life cycle**

Requirements completeness is judged continually, based on the changing needs of the project and team

The requirements must guide the current activities of all team members at an acceptable risk level

How Much Detail is Enough?

No requirements specification is ever truly complete

There isn't enough time or resources available to write them all – *and you shouldn't have to anyway…*

Provide detail where it's needed most: risky, unprecedented, or complex features and usages

Writing hundreds of pages of documentation may feel like productivity, but:

- If what gets documented is what everyone already understood, what is the effect on project risk?
- Large specifications can lead to a false sense of security

Make a conscious decision on what *NOT* to write

BRUF Versus Agile

Big Requirements Up Front (BRUF) involves asking stakeholders for "all their requirements", then "freezing" the requirements before design and development begins

BRUF forces stakeholders to defensively protect their interests by stating *every possible requirement they can think of*, even if it is unlikely they will ever need some of them

It is unreasonable to expect people to foresee all the contingencies and challenges up front

"Any attempt to formulate all possible requirements at the start of a project will fail and would cause considerable delays." Pahl and Beitz, *Engineering Design: A Systematic Approach*

Make a conscious decision on *WHEN* to write what you do write

A Flexible Approach to Scope and Details

Regardless of what type of system you are building, use an *evolutionary* approach to requirements engineering:

1. Start by generating requirements that define the scope of the system – full breadth, but minimum depth
2. Decide ***what not to write***
3. Decide ***when to write*** what you will write
4. Create the necessary details at the right time, always using business value and risk reduction as guides
5. Revisit steps 2 and 3 often based on what you learn as you make progress and the requirements evolve

Iterative and incremental work in an agile environment

The Three-Circle Model

Three Fundamental Perspectives

The **best** platforms and products...

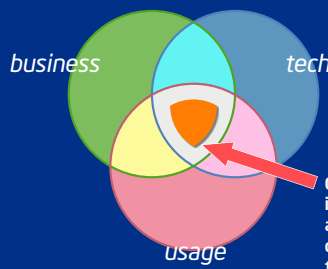
...are ***marketable*** and ***profitable*** → *business*

...are ***desirable***, ***useful***, and ***usable*** → *usage*

...are ***manufacturable*** and ***consumable*** → *technology*

The Three-Circle Model

The ***three*** circles combine to create ***seven*** regions



Compelling, integrated systems are found in the center, balancing all three perspectives

Balanced System Development

Although the three circles in the model are shown at the same size, there is a need for balance, not necessarily equality

Each new system presents its own challenges, as does the environment surrounding the system, the experience of the development team, and many other factors

Weighting business, usage, and technology perspectives according to these factors makes sense; ignoring a perspective does not

We need to develop systems using a balanced, systematic approach

Two-Circle Relationships

Value relates business and usage.

This interaction defines how usage contributes to market share, competitive advantage, and positioning

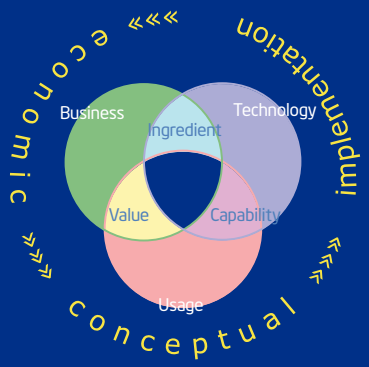
Capability relates usage and technology.

This interaction defines the interplay between usage, platform architecture, and supporting technologies

Ingredient relates technology and business.

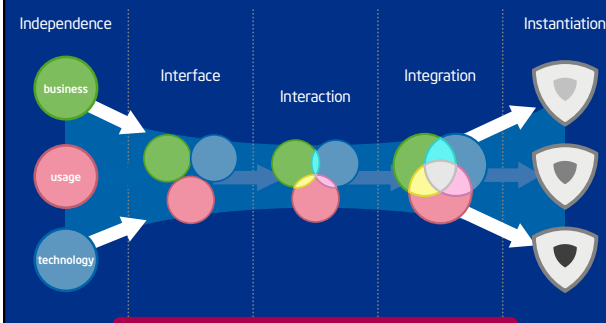
This interaction defines how technologies drive profitability and marketability

The Three-Circle Model Regions

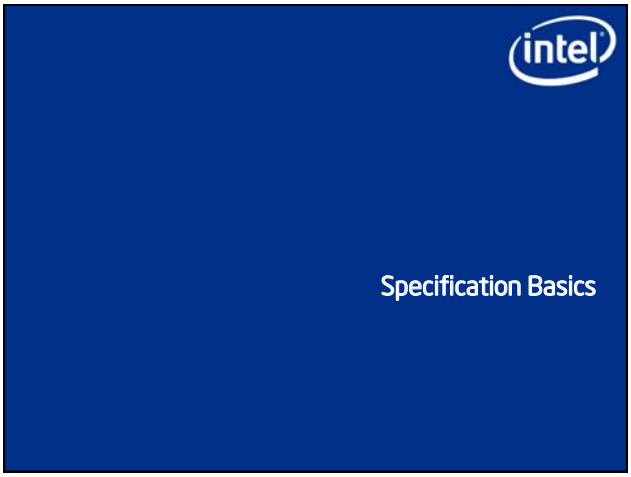


Integrating business, usage, and technology combines *ingredients* to provide a *capability* that delivers *value*

System Emergence



Systems emerge as business, usage, and technology perspectives converge



Specification Basics

These basic practices have a high return on investment:

- **Use a template** for requirements specification
- **Move from unconstrained natural language to constrained natural language** to reduce ambiguity and improve completeness with minimal effort
- **Do not include design statements** in the requirements unless they are there as intentionally-imposed constraints
- **Supplement natural language where needed** with other representations to improve comprehension and reduce ambiguity

Specification Basics, cont.

- **Quantify qualitative requirements** so they are verifiable
- **Define terms early and centrally** to ensure accurate use throughout the project
- **Validate requirements with stakeholders frequently** as a test of understanding
- **Rigorously review and inspect requirements** to prevent defects and maximize requirements quality

Attributes of a Good Requirement

- **Complete:** A requirement is complete when it contains sufficient detail for those that use it to guide their work
- **Correct:** A requirement is correct when it is error-free
- **Concise:** A requirement is concise when it contains just the necessary information, expressed in as few words as possible
- **Feasible:** A requirement is feasible if there is at least one design and implementation for it
- **Necessary:** A Requirement is necessary when it:
  - Is included to be market competitive
  - Can be traced to a stakeholder need
  - Establishes a new product differentiator or usage model
  - Is dictated by business strategy, roadmaps, or sustainability

Attributes of a Good Requirement

- **Prioritized:** A requirement is prioritized when it is ranked or ordered according to its importance
- **Unambiguous:** A requirement is unambiguous when it possesses a single interpretation
- **Verifiable:** A requirement is verifiable if it can be proved that the requirement was correctly implemented
- **Consistent:** A requirement is consistent when it does not conflict with any other requirements at any level
- **Traceable:** A requirement is traceable if it is uniquely and persistently identified with a Tag

Requirements vs. Design

"Requirements are the *what*, design is the *how*..."

This is true – to a point, but the main difference between requirement and design is one of perspective:

How you look at a statement dictates whether it is a requirement or a design

Executive management: *"A design to meet financial goals"* → Build a media center PC ← Product development: *"My requirements for this year"*

Requirements vs. Design

It's not whether a statement is a "requirement" or a "design" that matters, but whether the statement places *appropriate constraints* on the people that will read it

Many products carry the majority of their specifications forward from previous versions

If the system must be or act a certain way, say so...
If not, leave the people downstream as much freedom to do their jobs as possible

Using Imperatives

Use Shall or Must to indicate requirements

Either imperative is fine, but there is a traditional use of the two terms:

- **Shall** – Used in functional requirements
- **Must** – Used in quality and performance requirements

Should and **May** are not used for requirements, but may specify design goals or options that will not be validated

Will and **Responsible for** are not used for requirements, but may be used to refer to external systems or subsystems for informational purposes

Use of Should or May in a requirement often points to a missing trigger or condition

Negative Specification

It is appropriate to state what the system shall not do, but keep in mind that *the system shall not do much more than it shall do*

- Use negative specification sparingly, for emphasis
- Don't use negative specification for requirements that could be stated in the positive
- Avoid double negatives altogether

NO: "Users shall not be prevented from deleting data they have entered"

YES: "The system shall allow users to delete data they have entered"

Writing Functional Requirements

An excellent way to structure functional requirements is to use the following generic syntax:

[Trigger] [Precondition] Actor Action [Object]

Example:

When an Order is shipped and Order Terms are not "Prepaid", the system shall create an Invoice.

- Trigger: *When an Order is shipped*
- Precondition: *Order Terms are not "Prepaid"*
- Actor: *the system*
- Action: *create*
- Object: *an Invoice*

Writing Functional Requirements: EARS

A recent refinement of the generic syntax is the ***Easy Approach to Requirements Syntax*** (EARS) that contains patterns for specific types of functional requirements

Pattern Name	Pattern
Ubiquitous	The <system name> shall <system response>
Event-Driven	WHEN <trigger> <optional precondition> the <system name> shall <system response>
Unwanted Behavior	IF <unwanted condition or event>, THEN the <system name> shall <system response>
State-Driven	WHILE <system state>, the <system name> shall <system response>
Optional Feature	WHERE <feature is included>, the <system name> shall <system response>
Complex	(combinations of the above patterns)

Examples of Functional Requirement Syntax

The system shall allow the user to select a custom wallpaper for the display from any of the image files stored on the device.

When a user commands installation of an Application that accesses Communications Functions, the system shall prompt the user to acknowledge the access and agree before continuing installation.

When the system detects the user's face in proximity to the display while the phone function is active and Speaker Mode is off, the system shall turn off the display and deactivate the display's touch sensitivity.

While in Standby, if the battery capacity falls below 5% remaining, the system shall change the LED to flashing red.

What is Planguage?

Planguage is an informal, but structured, keyword-driven planning language

It can be used to create all types of requirements

The name Planguage is a combination of the words *Plan*ning and Lan*guage*

Planguage is an example of a Constrained Natural Language

Planguage aids communication about complex ideas

Planguage

Planguage provides a rich specification of requirements that results in:

- Fewer omissions in requirements
- Reduced ambiguity and increased readability
- Early evidence of feasibility and testability
- Increased requirements reuse
- Effective priority management
- Better, easier decision making

Beyond requirements, Planguage has many additional uses including success criteria, roadmaps, and design documents

Choosing Planguage Keywords

Requirements generally fall into two categories based on the nature of how they are measured:

Requirements measured in Boolean terms as either present or absent in the completed system

- This category includes *system functions* and *constraints*

Requirements measured on some scale or interval, as more or less rather than present or absent

- This category includes *system qualities* and *performance levels*, often also referred to as "non-functional requirements"

Because of the way they are measured, qualities and performance levels use some additional Planguage keywords

Basic Planguage Keywords for Any Requirement

ID: A unique, persistent identifier (often system-assigned)

Requirement: The text that details the requirement itself

Rationale: The reasoning that justifies the requirement

Priority: A rating of priority (numeric, HML, etc.)

Priority Reason: A short description of the requirement's claim on scarce resources; why is it rated as it is?

Tags: A set of keywords or phrases useful for sorting and searching

Stakeholders: a person or organization that influences a system's requirements or is impacted by that system

Basic Planguage Keywords for Any Requirement, cont.

Status: The status of the requirement (draft, committed, etc.)

Contact: The person who serves as a reference for the requirement

Author: The person that wrote the requirement

Revision: A version number for the statement

Date: The date of the most recent revision

Fuzzy concepts requiring more details: <*fuzzy concept*>

The source for any statement: ←

A Simple Planguage Requirement

ID: Invoice ← Christine Walsh

Requirement: When an Order is shipped and Order Terms are not "Prepaid", the system shall create an Invoice.

Rationale: Task automation decreases error rate, reduces effort per order. Meets corporate business principle for accounts receivable.

Priority: High. If not implemented, it will cause business process reengineering and reduce program ROI by $400K per year.

Stakeholders: Shipping, finance

Contact: Atul Gupta

Author, Revision, Date: Julie English, rev 1.0, 5 Oct 05

Additional Keywords for Quality and Performance Requirements

Ambition: A description of the goal of the requirement (this replaces the Requirement keyword used in functional requirements)

Scale: The scale of measure used to quantify the statement

Meter: The process or device used to establish location on a Scale

Minimum: The minimum level required to avoid political, financial, or other type of failure

Target: The level at which good success can be claimed

Outstanding: A stretch goal if everything goes perfectly

Past: An expression of previous results for comparison

Trend: An historical range or extrapolation of data

Record: The best known achievement

Quantifying Learnability

ID: Learnable ←C. Smith

Ambition: Make the system easy to learn ← VP marketing

Rationale: Upcoming hiring reflected in business plans makes learnability for order entry a critical success factor for new offices

Scale: Average time required for a Novice to complete a 1-item order using only the online help system for assistance.

Meter: Measurements obtained on 100 Novices during user interface testing.

Minimum: No more than 7 minutes

Target: No more than 5 minutes

Outstanding: No more than 3 minutes

Past: 11 minutes ← Recent site statistics

Defined: Novice: A person with less than 6 months experience with Web applications and no prior exposure to our Website.

Using Qualifiers

Qualifiers are expressed within square braces [ ] and may be used with *any* keyword

- They allow for conditions and events to be described, adding specificity to a requirement
- They most often contain data on *where*, *when*, etc.

Example: Instead of

Past: 11 minutes ← Recent site statistics

We could write

Past: **[1st quarter average, all orders, all regions, new customers only]**
11 minutes ← Recent site statistics

The Landing Zone

The Landing Zone

A Landing Zone is a table that defines a "region" of success for a product or project

The rows of the table contain the subset of requirements that directly define success or failure (*not* all the requirements)

The columns of the table contain a range of performance levels; usually, a Landing Zone covers the range between great success (Outstanding) and failure avoidance (Minimum)

Landing Zones can be used in agile development to help define success of an iteration or Scrum sprint

Landing Zones focus attention on what will create success

Example Landing Zone

Requirement	Outstanding	Target	Minimum
Retail On Shelf	Nov 15th	Nov. 22nd	Dec 1st
Manufacturing Cost	$9.00	$10.00	$11.50
Peak Project Headcount	250	350	400
Markets at Launch	US, APAC, EMEA	US, APAC	US, APAC
Design Wins at Launch	40+	30+	20+
Total First Year Volume	125K	110K	95K

Landing Zone Usage

Landing Zones are useful for several things:

- **Gain explicit consensus** at the start of a project on the definition of success
- **Quantify the achievement levels required** as an input to feasibility and risk analysis
- **Drive tradeoff discussions** and decision making throughout the project
- **Monitor and communicate** product attribute status to decision forums and management during development

Landing Zone Usage

Landing Zones help clarify decision authority for a team:

Decisions that do not violate any row of the LZ are made by the team as a normal part of their work

- So long as the team meets all LZ rows, that is success

Any decision that would cause any LZ row to be violated requires ratification from a higher authority

- This would include falling below Minimum or a decision to pursue something beyond Outstanding

Landing Zones can be created for platforms, components, service offerings, user experiences, projects, etc.

Landing Zone Variants

One Landing Zone variant adds a fourth column to monitor the level that the engineering team has committed to deliver:

Requirement	Outstanding	Target	Minimum	Commit

Another version drops the Outstanding level and replaces it with a Kill Switch level that, if reached, triggers a review meeting to consider stopping the project:

Requirement	Target	Minimum	Kill Switch

Customize Landing Zone format and content to meet your needs

Placing Functions in a Landing Zone

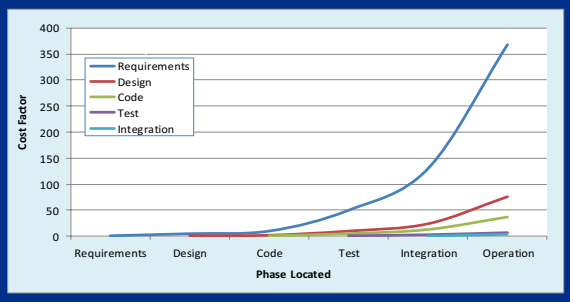
Landing Zone rows typically represent qualities and performance requirements that are measured across Minimum, Target, and Outstanding

Functions do not fit this pattern, but can be included in a Landing Zone by placement in a single row, where Minimum – Outstanding show different lists of functions:

Requirement	Outstanding	Target	Minimum
Retail On Shelf	Nov 15th	Nov. 22nd	Dec 1st
Functions	Target + HTML5 support	Min + Quad monitor, 4G	Dual monitor support, 3G

Specification Quality Control

Defect Removal Cost Relative to Phase Located



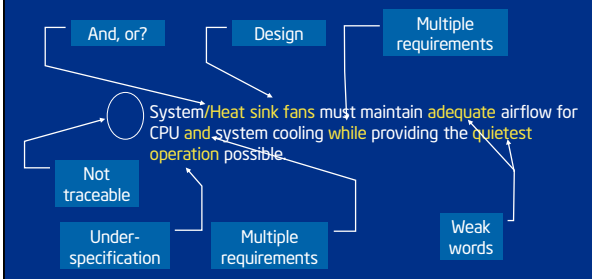
Source: NASA data, 2006

What's Wrong With My Requirements?

System/Heat sink fans must maintain adequate airflow for CPU and system cooling while providing the quietest operation possible.

See anything wrong?...

A Lot is Wrong, Actually…



Missing data: Source, status, rationale, priority, contact, etc.

Not verifiable as written

Peer Review Methods | Pros and Cons

	Pros	Cons
Informal Review	▪ Flexible ▪ Least threatening	▪ Finds fewer defects than other types ▪ Variable, inconsistent results
Walkthrough	▪ More systematic than reviews ▪ Identifies defects reviews miss	▪ May lack follow-up ▪ More time intensive and inconvenient than reviews
Inspection	▪ Most defects located ▪ Controlled, repeatable ▪ Industry proven practice	▪ Intimidating to some ▪ Requires training ▪ Can be too much effort without sampling

An Optimal Approach

An optimal requirements verification process would:

- Emphasize defect prevention and organizational learning
- Limit participant investment of time and energy to manageable levels
- Address the unique needs of each author and project
- Be suitable for all types and sizes of specification
- Rely on objective definitions and standards, not opinions
- Provide relevant, understandable metrics and indicators

The Answer: Specification Quality Control

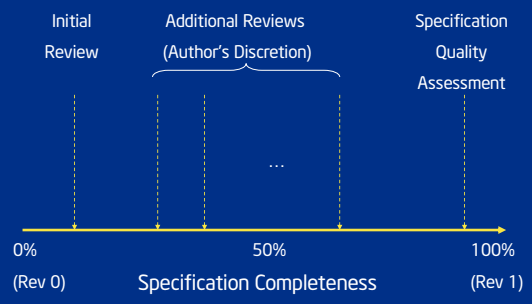
Specification Quality Control (SQC) is a method for ensuring *specifications* meet established quality goals according to objective, measured standards.

Specification Quality Control emphasizes:

- Cost and TTM reduction
- **Defect prevention**
- Resource efficiency
- Early learning
- Author confidentiality
- Quantified specification quality

Specification: Any representation (electronic or otherwise) of a requirement, constraint, design idea, plan, etc.

The Specification Quality Control Process



Why Specification Quality Control Works

Most requirements defects are repetitive, and can be prevented

- Early review allows an author to get timely, independent feedback on individual tendencies and errors
- By applying early learning to the rest (~90%) of the specification process, many defects are prevented before they occur
- This reduces rework in both the specification under review and all downstream derivative work products
- Over time, entire classes of defect are eliminated

Every time we have used SQC, requirements defect density has gone down by at least 50% – with only a few hours invested

Sample SQC Results

- A team using Scrum reduced requirements defect density from **15** major defects per 600 words in one sprint to **4.5** in the next
- A security technology team reduced defect density from **35** major defects per 600 words to **15** on their first attempt, then went on to achieve less than **5** within another 12 months
- A large software team reduced defect density according to the following table:

Rev.	# of Defects	# of Pages	Defects/ Page (DPP)	% Change in DPP
0.3	312	31	10.06	
0.5	209	44	4.75	-53%
0.6	247	60	4.12	-13%
0.7	114	33	3.45	-16%
0.8	45	38	1.18	-66%
1.0	10	45	0.22	-81%
Overall % change in DPP revision 0.3 to 1.0:				**-98%**

Summary

Requirements Engineering in the early 21st Century

Requirements engineering is changing based on complexity, pace, consumer choice, and similar factors

Agility, hierarchy, and abstraction will be key to success in developing complex future systems

Adopting a systems engineering-based perspective helps ensure appropriate focus on emergent behaviors and cross-cutting concerns

Several pragmatic, simple requirements practices fit this environment well: EARS, Planguage, Landing Zones, and SQC are good examples

Questions?

Thank You!

Sources for More Information

Software Requirements (2nd ed.), Karl E. Wiegers, MS Press 2003

More About Software Requirements, Karl. E. Wiegers, MS Press 2005

Competitive Engineering, Tom Gilb, Elsevier 2005

Software & Systems Requirements Engineering: In Practice, Brian Barenbach et al, McGraw Hill 2009

Just Enough Requirements Management, Al Davis, Dorset House 2005

Requirements Engineering: From system goals to UML models to software specifications, Axel van Lamsweerde, Wiley 2009

Software Requirements – Styles and Techniques (2nd ed.), Søren Lauesen, Addison Wesley 2001

Customer-Centered Products, Creating Successful Products through Smart Requirements Management, Ivy Hooks and Kristin A. Farry, Amacom, 2001

Sources for More Information

Requirements Engineering: Processes and Techniques, Gerald Kotonya and Ian Sommerville, Wiley 1999

Exploring Requirements: Quality Before Design, Donald Gause and Gerald Weinberg, Dorset House 1988

Effective Requirements Practices, Ralph Young, Addison Wesley 2001

Managing Software Requirements: A Unified Approach (2nd ed.), Dean Leffingwell and Don Widrig, Addison Wesley 2003

Non-Functional Requirements in Software Engineering, Lawrence Chung et al., Kluwer Academic Publishers 2000

System and Software Requirements Engineering (2nd Ed.), Richard H. Thayer and Merlin Dorfman (ed), IEEE 1997

Mastering the Requirements Process (2nd Ed.), James and Suzanne Robertson, Addison Wesley 1999

Backup

Some Additional Planguage Keywords

Gist: A brief summary of the requirement or area addressed

Assumptions: All assumptions or assertions that could cause problems if untrue now or later

Risks: Anything that could cause malfunction, delay, or other negative impacts on expected results

Defined: The definition of a term (better to use a glossary)

Wish: A desirable level of achievement that may not be attainable through available means

Kill Switch: A level at which the project would be cancelled or the product withdrawn from the market

{*item1, item2, ...***}** A collection of objects

See *Competitive Engineering* by Tom Gilb, or visit www.gilb.com for a complete list of keywords

Planguage Synonyms

The default version of Planguage as created by Tom Gilb uses different terms for a few of the keywords than Intel:

Default Terms		Intel's Terms
Must	→	Minimum
Plan	→	Target
Stretch	→	Outstanding
Catastrophe	→	Kill Switch
Tag	→	ID

Rather than use Tag as the unique ID for a requirement, Intel uses Tags to capture keywords or phrases used to search and sort, in keeping with common social networking use of the term

Example of Early Planguage Use (1988)

Usability Attribute	Measuring Technique	Metric	Worst-Case Level	Planned Level	Best-Case Level
Initial use	NOTES benchmark task	Number of successful interactions in 30 minutes	1-2	3-4	8-10
Initial evaluation	Attitude questionnaire	Evaluation score (0 to 100)	50	67	83
Error recovery	Critical-incident analysis	Percent incidents "covered"	10%	50%	100%

This table comes from a Usability Specification written by DEC in 1988 for VAX NOTES Version 1.0. It bears a striking resemblance to a landing zone.

Examples of Scales and Meters

Tag: Environmental Noise
Scale: dBA at 1 meter
Meter: Lab measurements performed according to a <standard environmental test process>

Tag: Software Security
Scale: Time required to break into the system
Meter: An attempt by a team of experts to break into the system using commonly available tools

Tag: Software Maintainability
Scale: Average engineering time from report to closure of defects
Meter: Analysis of 30 consecutive defects reported and corrected during product development

Examples of Scales and Meters

Tag: System Reliability
Scale: The time at which 10% of the systems have experienced a <failure>
Meter: Highly-Accelerated System Test (HAST) performed on a sample from early production

Tag: Revenue
Scale: Total sales in US$
Meter: Quarterly 10Q reporting to SEC

Tag: Market
Scale: Percentage of Total Available Market (TAM)
Meter: Quarterly market surveys

Remember: Scale = units of measure,
Meter = Device or process to measure position on the Scale

Liftoff: Starting New Projects on a Trajectory Toward Success

Diana Larsen, FutureWorks Consulting

Liftoff – it's the unexplored, often ignored, Agile software development project practice. Liftoff gives impetus to your projects in a way that starts the project team, and the business, on the trajectory to success. Project sponsors, product managers, and product owners use these critical meetings to inform, inspire, and align the people who do the work with the definition of the work to be done.

As the first act of the flight, a rocket launch requires an entire set of systems to successfully lift the vehicle into orbit – not just the vehicle itself, but all the systems needed for smoothly moving off the ground into space. Likewise, your project needs its entire set of supporting systems in place to begin a successful journey to high performance, hyper-productivity, and high value delivery.

In this talk, Diana Larsen explores ways to accomplish Liftoff, including the vital step of chartering the project. She'll share real-life stories of how others have effectively started their projects; a variety of team activities to fuel your Liftoff; and a framework for effective, "just enough" Agile chartering.

Drawing on 20+ years of experience working with technical professionals, Diana Larsen takes a pragmatic approach to consulting with leaders and teams to promote workplaces where innovation, inspiration, and imagination flourish.

As partner and senior consultant with FutureWorks Consulting, Diana Larsen sparks the creation of workplaces where productive teams display resilience in times of change and focus on frequent delivery of high value software customers want and use. She leads system-wide groups in collaborative thinking and planning for agile adoptions, project kick-offs, chartering, and retrospectives. Diana coaches managers and leaders on their role in a changing workplace and presents workshops on topics related to agile transitions and teams. She also directs the Agile Adoption program of the Agile Alliance and serves on the Agile Alliance Board of Directors. Diana co-authored Agile Retrospectives: Making Good Teams Great!

Liftoff

Starting New Projects
on the Trajectory toward Success

Diana Larsen
co-author, *Liftoff: Launching Agile Teams and Projects toward Success*
http://futureworksconsulting.com

PNSQC 2011

Success = High Value Delivery

What the customer wants and values

That creates value for the business

That the customer will accept & exchange value for

In a timeframe that suits the customers' needs

Easily maintainable and supportable after deployment

In a way that leaves team members ready and eager to work on the next deliverable

What do you do to help project teams get off to a good start?

Start with a Booster

Project Kick-Off Workshop

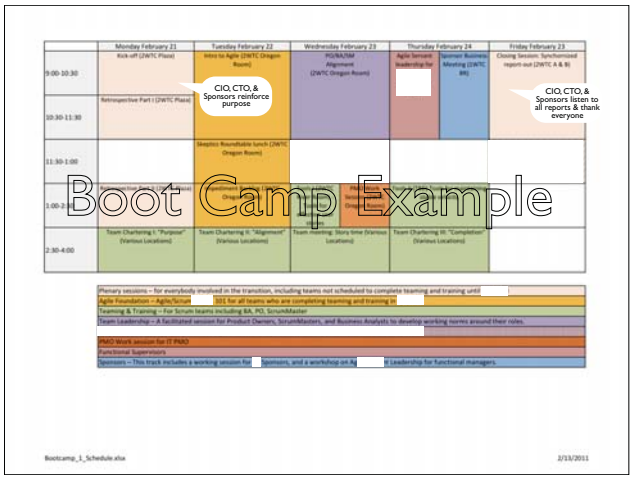
"At PSE, the software development division of Siemens AG Austria, we have realized significant benefits by providing carefully designed facilitated rituals at the beginning and at the end of projects, called kick-off workshops and project retrospectives respectively. The high return-on-investment for the time spent is unchallenged and confirmed by the regular feedback of developers, project leaders, and managers. They attest a positive influence of these rituals on many subjects such as team cooperation, process effectiveness, quality assurance, know-how sharing, reliability of estimations, and so on. This helps to improve the development cycle and increases the financial success of our projects."

Frowin Fatjak (PSE, Siemens AG Österreich),
"Kick-off Workshops and Project Retrospectives:
A Good Learning Software Organization Practice"
Wissensmanagement (LNCS Volume) 2005

"The term kick-off designates an internal workshop at the beginning of a project or at the start of a project phase. A kick-off at PSE usually lasts for 1-2 days.

Features:
Whole Team participates and contributes
Agenda with variety of activities, presentations by different people, & moderated discussions
Higher awareness of and attention to risks

Success Factors:
Facilitator/Leader contracting
Establishing Trust
Whole Team involvement
Recording Minutes (digital camera)
Offsite Location"



Plan to Promote Team Forming

Provide structure, information, and support
Draft an Agile Charter
Focus on the "do-able"
Define "done"
Identify roles and responsibilities
Initiate iteration retrospectives
Acknowledge feelings of newness or confusion
Manage participation so everyone has a voice
Set a tone of openness and trust

Project Liftoff Activities

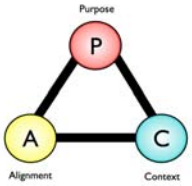
Collaborative Chartering
Kick-Off Workshops
Iteration 0
Boot Camps
Retrospective & FutureSpective
Open Space

Which of these have you tried?

Agile Chartering

Chartering the Product Team

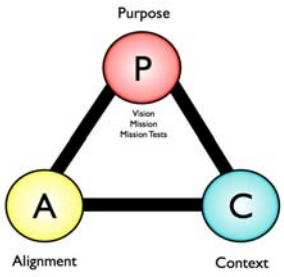
Inspiration & Meaning



Create an alliance

Understand fit with organization

Living Charter = Chartering



Vision - Value to Attain

Mission - Result to Accomplish

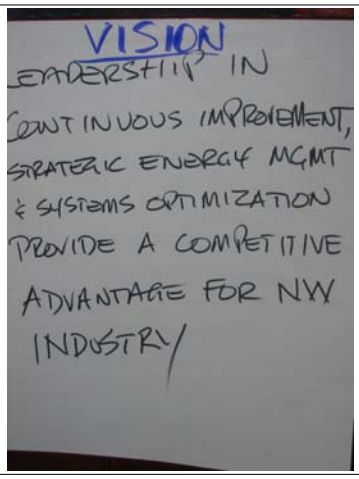
Mission Tests - Criteria for Success

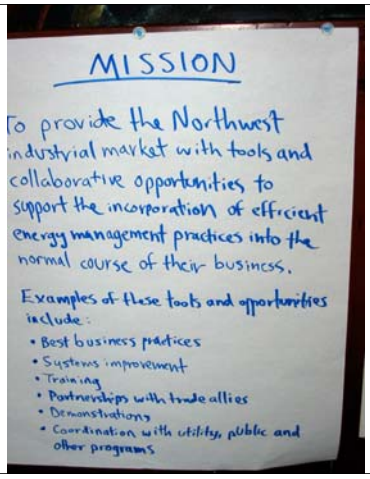
"TinCans" Purpose

- **Vision** TinCans makes long distance collaboration easier by eliminating the bottlenecks of previous desktop sharing tools. TinCans is full of the small touches that customers find joy in using, and is known for collaboration, simplicity and high quality. With the creation of TinCans, we deliver our first entrepreneurial product. The successful launch of Tin Cans establishes us as a strong competitor in the collaboration tools marketplace.
- **Mission** TinCans is a freeform sandbox that fulfills multiple purposes, such as: project management, working in pairs & subgroups, project tracking, distributed retrospectives, and more. TinCans helps teams collaborate over long distance, and enables the productive team dynamics that occur when teams gather around a table and use index cards to brainstorm, prioritize, and reflect. Collaboration, simplicity, and high quality take precedence over breadth of features, and we focus on polishing existing features to a high gloss before adding new ones.
- **Mission Tests** We will deploy TinCans in multiple stages with increasing measures of success at each stage:
  - By Q2 2009, we will demonstrate a proof-of-concept at a major Agile event garnering a positive response from Agile experts for its simplicity and high quality.
  - By Q3 2009, we will make a TinCans beta available for free use and within 6 months at least 10 teams will use it on a regular basis for real-world projects.
  - By Q2 2010, TinCan will convert to SaaS and gross at least $10K in the first 3 months.
  - By Q2 2011, we will rely on TinCans for our income and the revenues will meet our monthly minimum income requirements.

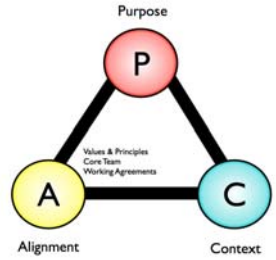
"Racoon Rescue Website Project" Purpose

- **Product Vision** Racoon Rescue enables residents, outdoor enthusiasts, and local communities to live in harmony with regional native wildlife and preserves wildlife habitat for future generations. Its website attracts volunteers and donations that sustain this critical work.
- **Project Mission** The Racoon Rescue Website project provides a way to disseminate information about RR; attract and register volunteers and supporters; and receive donations.
- **Mission Tests** We will deploy the RR website in multiple stages with measures of success at each stage:
  - By Q4 2011, the site will have processed 20 troublefree donations.
  - By Q1 2012, the pool of volunteer registered through the website will increase by 25%.
  - By Q1 2012 RR brochure requests increase by 50%, either through mailing triggered by website registrations or reports that website influenced the request.





What's your current project purpose?



Values & Principles - Beliefs & Ideals about Work

Core Team - Cross-functional group with a common purpose.

Working Agreements - Operational Guidelines

"Tin Cans" Alignment

- **Values & Principles** -
  Openness: Openness is not an end in itself, but a means to an end. We encourage everyone to speak up on topics that concern them or the team as a whole.
  Outcome-Focus: We evaluate work products not the author.
  Feedback: Because we want the highest quality working relationships, we encourage, welcome, and seek effective feedback. We give feedback with caring and respect.
  Innovation: We encourage each other to try new ideas. We regard mistakes not as failure, but as experiments on the road to success.
- **Core Team -** Membership: Oanh, Phil, Quincy, Ruth, Siraj, Ted, Erik, and Uma. Individual Contributions & Work Styles. Interdependent Skills. Cross-functional work. Each person's PWTP (project within the project). Laters.
- **Working Agreements -**

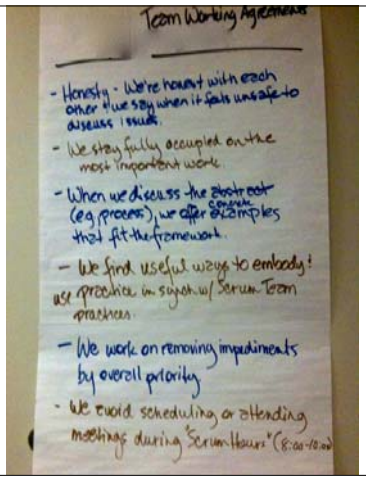
As a team, we believe we work best when:
We define "done" for our products as "ready to be accepted and user by our customer.
We hold planning meetings before and retrospectives after each iteration and release.
We value curiosity; therefore, we listen and we encourage each other to express ideas.
We seek opportunities for continuous learning and sharing knowledge.
We find ways to make our production work fun for the whole team.

"Racoon Rescue Website Project" Purpose

⋆ **Values & Principles** We value:
Sustainability - Make decisions with the long term in mind.
Relationships - Give and expect respect; all living beings together create a living planet.
Beauty - Seek the highest quality and beauty in all we do.
Transparency - Share information freely.
Courage - Speak up when something needs to change.

⋆ **Core Team** Website team members include: product owner, graphic designer, copy writer, volunteer coordinator, user experience specialist, web programmer, database programmer, and tester. We will reach out to others for expertise in database administration, security, financial transactions, etc.

⋆ **Working Agreements** We work together best when:
* Every team member completes work as agreed and let's the team know if anything gets in the way of completing a task.
* For us, "done" means designed, coded, re-factored, passed unit & acceptance tests, integrated, ready for deployment to internal server, and accepted by the product owner.
* We hold brief daily meetings at a time when everyone can attend, and we all show up in person or by call in.



Sample Working Agreements for Cultivating Trust

We agree to assume positive intent and give generous interpretations to actions or words we don't understand, then we seek clarity from one another.

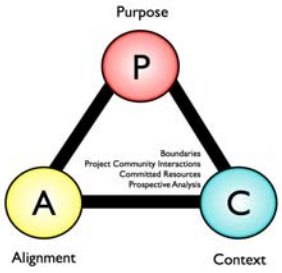
We keep our agreements or, if we can't, we advise teammates of problems as soon as possible.

We cast no "silent vetos." We speak up if we disagree.

We seek and offer feedback on the impact of our actions, inactions, and interactions.

28

What's one working agreement that would help your team work better together?



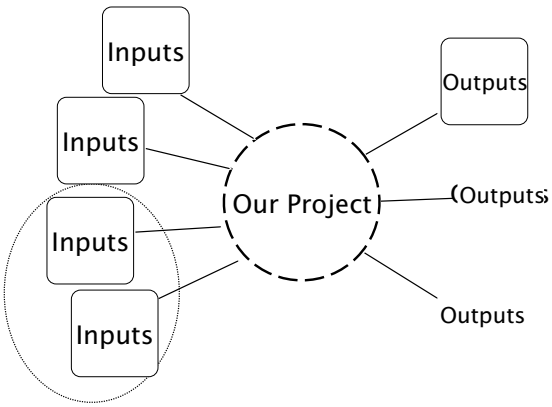
Boundaries & Interactions - Seeing the Systems

Committed Resources - Organization Support

Prospective Analysis - Initial Projections

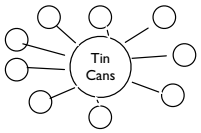
Example: Boundaries & Interactions for Writing a Book (Liftoff)

Project boundaries included:
scope (content, book writing may include related articles)
timeboxes (4-6 day pair writing blocks)
project boundary objects (see right)
limits of authority (each author must agree in advance to public statements about the work)



"Tin Cans" Context

- ⋆ **Boundaries & Interactions** Tin Cans boundaries include scope, timeboxes (1 week iterations, 2 month releases to internal production), project boundary objects (see below) project community members' relative responsibilities, and limits of authority (purchasing up to $10K, recruit 1 additional programmer, all decisions about work hours within a 40-hour week,).



- ⋆ **Committed Resources** Tin Cans sponsors agree to provide a facilities budget for creating an open work area, including two build servers, and access to industry analysts and reports for our market. Core Team members may schedule 4 hours per week to work on personal projects.
- ⋆ **Prospective Analysis** We've identified a moderate probability, high negative impact risk others may be building a similar product and beat us to the market. Mitigate by prioritizing "first to market" in feature and release decisions.

Activity: Identifying Boundaries & Interactions

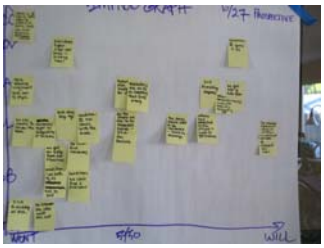
- In groups of 3 or 4, review the purpose of the Racoon Rescue Website Project.
- Identify and discuss the significant boundaries of the project
- Identify members of the project community
- Create a context map
- Label exchanges between team and the project community parties

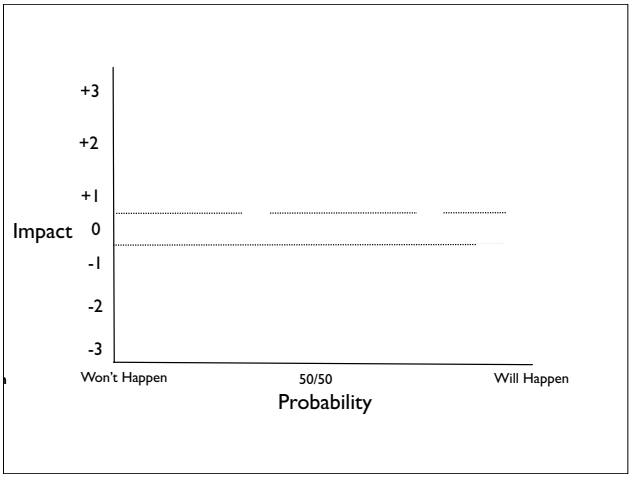
"Racoon Rescue Website Project" Purpose

- ⋆ **Product Vision** Racoon Rescue enables residents, outdoor enthusiasts, and local communities to live in harmony with regional native wildlife and preserves wildlife habitat for future generations. Its website attracts volunteers and donations that sustain this critical work.
- ⋆ **Project Mission** The Racoon Rescue Website project provides a way to disseminate information about RR; attract and register volunteers and supporters; and receive donations.
- ⋆ **Mission Tests** We will deploy the RR website in multiple stages with measures of success at each stage:
  - * By Q4 2011, the site will have processed 20 troublefree donations.
  - * By Q1 2012, the pool of volunteer registered through the website will increase by 25%.
  - * By Q1 2012 RR brochure requests increase by 50%, either through mailing triggered by website registrations or reports that website influenced the request.

Example: More "Book Writing" Context

- ⋆ **Committed Resources**
  - Co-located time together – min. ~1 week / quarter & min. 2 phone check-ins/mo.
  - Alternate travel to visit each other - at least two trips per year
  - Split printing & shipping $ as they come up
  - Each will buy on our own any book purchased for research that we want to keep
- ⋆ **Prospective Analysis**



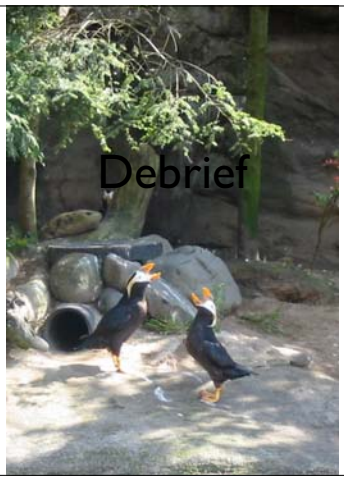


Activity: Creating a Prospective Analysis

- In groups of 3 or 4, review the purpose of the Racoon Rescue Website Project.
- Using sticky notes, in 3 minutes brainstorm at least a dozen conditions that may affect the project in the next 3-4 months
- Create a graph at your table, and place each sticky note according to its impact and likelihood
- Discuss the patterns you see

"Racoon Rescue Website Project" Purpose

- **Product Vision** Racoon Rescue enables residents, outdoor enthusiasts, and local communities to live in harmony with regional native wildlife and preserves wildlife habitat for future generations. Its website attracts volunteers and donations that sustain this critical work.
- **Project Mission** The Racoon Rescue Website project provides a way to disseminate information about RR; attract and register volunteers and supporters; and receive donations.
- **Mission Tests** We will deploy the RR website in multiple stages with measures of success at each stage:
  - By Q4 2011, the site will have processed 20 troublefree donations.
  - By Q1 2012, the pool of volunteer registered through the website will increase by 25%.
  - By Q1 2012 RR brochure requests increase by 50%, either through mailing triggered by website registrations or reports that website influenced the request.



What difference would it make to understand more about project context from the beginning?

Coming Soon!

Liftoff: Launching Agile Teams and Projects Toward Success

Get in touch:

Email dlarsen@futureworksconsulting.com

URL http://futureworksconsulting.com

Blog http://futureworksconsulting.com/blog

Twitter DianaOfPortland

Linked In http://www.linkedin.com/in/dianalarsenagileswd

How to Deal with an Abrasive Boss (aka "Bully")

Pam Rechel, Brave Heart Consulting

pam@braveheartconsulting.com

There are bosses who are not as competent as we would like, some who are annoying but tolerable and some who are abrasive. Abrasive bosses (sometimes known as "bullies") behave in such a negative manner that the workplace is impacted – motivation is low or nonexistent, results begin to slide downward and employees hunker down to stay out of the way or begin their exit strategy. Employees, who don't have authority over the boss, are often so afraid of the abrasive boss or not wanting to trigger a reaction, that nothing is said or done.

In the session we'll discuss why people behave in abrasive ways; some strategies that can work to deal with them and what strategies to avoid. The goal of this session is to present practical steps on how to deal with an abrasive boss.

It's important to take control of your work life and to be able to change how you approach someone whose behavior is unacceptable. The skill of communicating with difficult individuals is often not taught by our parents or schools, yet it becomes a key workplace competence to survive and thrive at work.

By the end of the session, participants will learn two strategies to successfully deal with workplace bullies and two strategies to avoid.

Pam Rechel is an executive coach and organization consultant with Brave Heart Consulting in Portland specializing in coaching abrasive managers and helping busy teams increase accountability and results. She has an M.A. in Coaching and Consulting in Organizations from the Leadership Institute of Seattle (LIOS), an M.B.A. in Information Systems from George Washington University and an M.S. from Syracuse University and is a Newfield Network certified coach.

Before We Can Test, We Must Talk: Creating a Culture of Learning Within Quality Teams

Kristina Sitarski
Menlo Innovations
E-mail: kristi.sitarski@gmail.com

Abstract:

Learning is a process you do, not a process that is done to you. Before the heated discussion on what automated functional testing tool to lobby for or the fun game of 'who can cause the exception first,' we must learn how to communicate with one another. In my experience working in a paired quality assurance environment for three years, I have been partnered with several different people. Although my partners have been problem-solving software destructors like myself, they all have had very distinct learning styles. This brings me to a truth sometimes overlooked: to teach is to learn. To help teach not only my partner, but also customers and our Agile shop tour attendees, I look to Neil Fleming's model of different learning styles. Using this as a guideline, as well as some important items ranging from jellybeans to yarn, I find it has improved the communication on concepts such as functional test writing, object modeling, and estimation.

Understanding Mr. Fleming's model is important, but it is imperative to constantly be self aware of the audience you are communicating with and not afraid to create your own teaching style.To deliver quality communication, sometimes on quality itself, we must learn how to effectively teach and learn from one another.

Biography:

Kristina Sitarski has worked at Menlo Innovations as a quality advocate since 2008. Before she became a quality advocate, she worked in quality assurance for Xerox. Currently at Menlo, she brings her experience, as well as a degree in anthropology, to provide an objective, user-centric view of the design and implementation of software tests. She enjoys working and living in the city of Ann Arbor, Michigan.

1. Introduction

Before we can define and deliver quality communication, we must first learn how to effectively teach one another. Effective teaching skills are a concept that can be applied to almost all facets of life, but when given the opportunity to apply it in the workplace (in this case, software testing) it becomes clear there is a dramatic effect on the quality of tests produced as well as the enjoyment level of the testers. I have been given the opportunity to refine and apply effective teaching techniques while working at Menlo Innovations because it is a software company that has developed a culture based on the Agile methodology and human factored design. It is obvious that effective communication is an essential part of this culture, since we practice not only paired programming but also paired testing. It is a common phrase to hear around the factory and among pairs, “make mistakes faster.” To Menlonians, this translates to: making more mistakes gives more opportunities for learning and discussion. The more active discussions we have the faster our knowledge base grows as a team. The “make mistakes” part is easy, but making the discussions meaningful can sometimes be difficult. Like many things in life, the hard part is where we gain the value. These constant discussions are a part of life at Menlo because you are partnered with someone for eight hours of the day. It is the most dynamic collaboration I have ever experienced.

2. A Paradigm Shift

Before my time at Menlo, the whole idea of quality assurance seemed very black and white to me and the words “dynamic collaboration” did not fit into the picture at all. My experience in the QA world to that point had been mostly reading specification documents and following a process. As far as my definition of communication went, a simple one of sending and receiving messages applied nicely. I would receive an e-mail message from my manager, figure out the problem, and send an e-mail back. My manager sat five feet away and yet we rarely spoke. I would rate my performance by the number of e-mails in my inbox. If I was down to twenty messages I felt I was doing a good job. I was also getting good at figuring out problems. In fact, I had figured out a pattern for finding certain types of issues. It never occurred to me to share this information with others. The whole concept of teaching seemed absurd, as I was being rewarded for my work and no one wanted to hear what I had to say. In that environment, speaking outside your realm of work was taken as a challenge to authority or as trying to steal someone’s job. At some point I decided I wanted a serious career or, in actuality, I was afraid I would die of boredom. Through that job, I did discover some important things about myself. I have a knack for finding faults in systems that are not obvious to others. With this new self-realization, I went to search for another job.

With a degree in anthropology, I was not overly confident about my interview with Menlo, a seven-year-old software company, but I would soon learn my father’s inquisitive ways with computers would help me more than I could ever imagine. I was never as much interested in the latest technology as he was, but when one has to step over carcasses of computers in order to get to the laundry room, I suppose one's natural curiosity takes over. I am reminded of the father from the movie, *The Goonies,* except instead of the “Bully Buster glove,” it was an MP3 digital stereo on my belt. As the date of my interview with Menlo drew closer, my nervousness turned to curiosity after receiving the invitation that had the description of “Extreme Interview.” What on earth could this mean? Was I going to be interviewed under some extreme circumstances? When I arrived, there was a group of about 20 people waiting around to be interviewed and, last time I checked, their website had listed only 2 positions open. Things got even weirder when I

was partnered with another person applying for the same job and asked to then, “help you partner get the job.” Are these people crazy? Have I accidentally walked into a speed dating event?

We sat down across from a current Menlo employee who was there to observe our “kindergarten skills” as we tried to perform simple problem solving exercises together. What seemed terrifying soon became refreshing. I was in a group of people that actually cared not just about having the problem solved, but about the *way* in which it was solved. Even though I was working with complete strangers, I remember feeling so creative and inspired that I forgot I was in an interview and eventually started thinking, “this is fun.” I am fortunate to say I still get to have fun and get paid. The uniqueness of that interview carried on into the everyday work environment, as I now have a partner who I work with every day. It’s just like all those famous cop shows, except instead of killers we are chasing down database errors. When I started my new QA job, I would use the application, find a bug, and look to my partner for a congratulations. Instead of congratulatory, the look on their face was perplexed and almost slightly annoyed. They would ask, “why did you click there? Why did you do that?”

I didn't know how to respond. I felt like I had suddenly forgotten the English language. Why couldn't I verbalize my train of thought? Could I just send them an e-mail explanation later? I quickly realized that my old model of communication was not going to work.

I tried my best to talk out my thought process as it was occurring, but it seemed like that only resulted in sentence fragments and rapid bursts of excitement. Finally, I found myself grabbing scraps of paper and drawing pictures. Most of the time these pictures made little sense to anyone but me, but the important part was that, somehow, if I had a squiggly line to point to, it would help the words come out of my mouth. It wasn't the picture, but the conversation it incited between me and my partner. As I had more practice at this, it became very important to make sure the other person understood why I was doing something. Sharing thoughts in a way that was effective and organized allowed my partner to contribute to the problem at hand in a more meaningful way. It was collaboration and it was fun. The next week, I was assigned to a new partner. I was now the senior in the pairing and it was no longer adequate to have the skill of being able to explain my thoughts regarding a specific test setup. I now had to explain high-level architecture and present testing plans to clients. Things were not going smoothly. We would discuss a plan of action, but then my partner would execute something completely different. As it turns out, my partner was confused so much by my meaningless squiggles and lines that he couldn't focus on what was being said in the conversation. He would agree to the test plan, but his plan was something different than what I thought we had just discussed. This was when I realized, just like with my first partner, we needed to figure out how to talk and start truly communicating.

3. Learning to Learn

I began researching Neil Fleming’s model of different learning styles. In this model, Fleming proposed that one could fall into three different categories of learning: visual, auditory or kinesthetic.

> Fleming claimed that visual learners have a preference for seeing or thinking in pictures, using visual aids such as overhead slides, diagrams, handouts. Auditory learners best learn through listening, lectures, discussions, or tapes. Tactile/kinesthetic learners prefer to learn via experience—moving, touching, and doing, active exploration of the world; science projects; experiments. (Learning Styles Explained 2008)

Fleming had even developed a questionnaire that would determine your style of learning. My partner and I were very excited to discover what category we fell into and we both took the survey. Fifteen minutes later, the results were in; I ranked highest in the kinesthetic category and lowest in the auditory. My partner ranked highest in the auditory and lowest in the kinesthetic. We were delighted by all this data and quickly went to the next page of the questionnaire in hopes that this is where we would find all the answers to our problems. Of course the next page read, “there are individualized learning profiles (pages of advice) available for purchase.”

What followed next was a discussion of what all these categories even mean and if we even care. What we took from this was the ability to be more self-aware while teaching. It was important to develop a way of communication in a style that was more likely to be understood rather than the style in which the teacher understands things. It is a very simple concept. Understanding it is the key to self-awareness. By researching the different learning styles I became more aware of when I was about to go off on a tangent; talking in metaphors and opening 500 different tables in the database. We exposed ourselves to different ways of teaching and decided that just one method of teaching would not work.

Menlo, as an organization, was more advanced in this regard than our small quality advocacy team. I started noticing in classes and in the tours that were offered to the public and new clients, how presenters geared their teaching towards all of the different styles of learning. A great example of this was an exercise that was developed to help a new team understand the definition of object oriented design and what an object model is. It was especially helpful during a time when Menlo was transitioning a project back to a development environment located at the client site. The clients had just hired a new team and were now ready to take over two years of code written at Menlo. Many of their programmers were getting frustrated at not understanding why the code was sometimes written in a way that seemed like more work in order to produce a fairly simple function. The client sponsors, who are non-technical people, were having a hard time understanding their team’s frustrations. To help solve this problem, we had a meeting involving both teams representing roles such as: developers, QA, project sponsors, and solution architects.

Step One: we divided into mixed groups of teams across roles.
Step Two: someone read aloud the brief definition of an object model from a reference:

> An object-oriented program will usually contain different types of objects, each type corresponding to a particular kind of complex data to be managed or perhaps to a real-world object or concept such as a bank account, a hockey player, or a bulldozer. A program might well contain multiple copies of each type of object, one for each of the real-world objects the program is dealing with. For instance, there could be one bank account object for each real-world account at a particular bank. Each copy of the bank account object would be alike in the methods it offers for manipulating or reading its data, but the data inside each object would differ reflecting the different history of each account. Objects can be thought of as wrapping their data within a set of functions designed to ensure that the data are used appropriately, and to assist in that use. (Kay 2003)

Step Three: making sense of the artifacts in front of us on the table, which included: stacks of different colored note cards, rulers, markers, and three different types of graphing paper.

The goal was to build a mock graphing calculator using object oriented design. Soon the room was full of discussion. What are the objects we need? What are the functions of those objects? How do they relate? Soon we had project managers writing Class-Responsibility-Collaboration (CRC) cards saying things like, “but why isn’t our data set object in charge of drawing?” It is important to note that not only did we decide

on what our objects would be, but we acted as if we were the graphing calculator ourselves. We were physically drawing the points and lines that were the output from our math equation object we had created. This triggered ideas that would have been lost if we simply had a real graphing calculator do the work, or worse, just conceptually thinking about what the calculator would do. It also gave an outlet for ideas such as, “What if our graphing calculator was voice activated? Can our lines draw in different colors?”

To wrap up, each group presented on what they had accomplished. In a matter of two hours we had used every learning style Fleming could have imagined and created a common language that was shared between technical and non-technical people. The team learned a style of collaboration that made for quality communication. The team also took away a better concept and appreciation for object oriented development and design.

4. Conclusion

Building and testing great software is hard work; talking should be easy. It is always difficult to start a cultural shift in thinking among teams, especially because it must come from the team members themselves. A manager cannot enforce a cultural process of good communication. The team must build the habit of talking aloud and with one another. They must support each other and know when to suggest moving the problem -solving away from the computer and to a white board. It is imperative to constantly be self-aware of what you are doing and why you are doing it. You should always be able to teach your partner your thought process and talk through why your testing steps are organized in the way they are. It is easy to suggest, “oh it’s just random” or “I just have a hunch it might break there,” but it is vital to talk through these steps more thoroughly, while keeping a person’s personal learning style in mind and constantly sharing information.

Constant communication not only fosters learning but creates a safe environment where people feel it is okay to share these small details and where questions can be asked that would normally be overlooked or deemed not important. In actuality, the smallest thought could be something that turns the light on or fills a missing link for someone else. Before you realize it, you are learning something completely new and a problem is on the right track for being solved rather than still being worked on alone for hours. The values that make a great teacher are similar to the values of a quality team: clear, written-out objectives, looking at issues in a variety of ways, having a strong sense of caring for the end user and, above all, a passion for learning.

References

1. “*What are learning styles?,*” LdPride. (n.d.)., accessed October 17, 2008, http://www.ldpride.net/learningstyles.MI.htm#Learning%20Styles%20Explained.

2. “Dr. Alan Kay on the Meaning of "Object-Oriented Programming",“ Kay, Allen PhD, 2003. accessed 2010-02-11, http://userpage.fu-berlin.de/~ram/pub/pub_jf47ht81Ht/doc_kay_oop_en

Delivering Quality One Weekend at a Time: Lessons Learned from Weekend Testing

Michael Larsen
larsenmsk@gmail.com

Abstract

For many testers, the steps taken to learn and grow our craft are often ad-hoc and random. We learn from books, we ask questions and receive answers on online forums and web sites, we attend conferences and confer with peers, and we may talk to fellow testers at work. But what if those resources are not enough?

What if we had an organization that testers could turn to, where they could, practice the craft of testing, and learn from mentors from all over the world? What if such a group would tackle interesting problems and encourage both novices and experienced testers to participate? What if I said such a group was available completely free of cost to participants? Sound like fantasy? It's not; it's happening here and now. The group that does this is called "Weekend Testing". While the concept of Weekend Testing has been around for a few years, the power is in the utilization of the methods and principles, not the name or even the people behind it.

The paper explains the methods that we use in Weekend Testing, and how you can participate in sessions that Weekend Testing chapters arrange. More important, even if you never participate in an official Weekend Testing session, you can apply the concepts and methods to your group or organization. I will explain how to facilitate a Weekend Testing Session, and share some lessons learned from dozens of Weekend Testing sessions over the past couple of years. Additionally, I will show you how to take these same ideas and use them with your own organizations.

Biography

Michael Larsen is Senior Tester at Sidereel.com in San Francisco, California. Over the past seventeen years, he has been involved in software testing for products ranging from network routers and switches, virtual machines, capacitance touch devices, video games and distributed database applications that service the legal and entertainment industries.

Michael is and co-founder and facilitator of the Americas chapter of Weekend Testing, an instructor with the Association for Software Testing, a Black Belt in the Miagi-Do School of Software Testing, and the producer of Software Test Professionals "This Week in Software Testing" podcast. He also contributed the chapter "Trading Money for Time" to the book "How to Reduce the Cost of Software Testing", scheduled to be released in October, 2011. Michael writes the TESTHEAD blog (http://mkl-testhead.blogspot.com) and can be found on Twitter at @mkltesthead.

1. Introduction

We all hope to find ways that we will be able to work with and mentor other testers. In some organizations that is already in place and may be able to cover certain testing needs, but there are times where this approach does not meet particular needs:

- What if you have a need to learn a new approach to testing?
- What if you are a lone tester and do not have the benefit of a dedicated test team to work with?
- What if you are a young tester who has heard about many of the test techniques written about, but have no experience using them?

Weekend Testing is a movement of individual testers from around the globe that have made it their mission to tackle these problems, and they do so voluntarily. Its mission is to help develop testing skills and encourage veteran testers to mentor younger or less experienced testers. WT is a place where testers

- Learn and practice in a fail-safe environment
- Have an opportunity to give back to the community
- Learn through collaboration
- Share experiences
- Learn about new tools and technologies
- Share and receive peer feedback

Weekend Testing has a dedicated site and is actively maintained at http://weekendtesting.com/

Weekend Testing was originally formed in Bangalore, India. Many of the active testers were looking for ways to learn more about testing and to get better at it. They realized that many of the scripted test methods that they had been originally taught were insufficient in light of the challenges they were facing. Additionally, they wanted to create an environment that would allow testers from all areas of experience to have a forum to participate, practice and learn from each other. Rather than look for canned options that didn't address the issues and problems they were seeing, they banded together to help teach and learn from each other. It was officially founded on Aug 1st, 2009. The first public session was held Aug 15th, 2009. This date was symbolic because it marks Indian Independence day; the founders felt that they were likewise giving “freedom” to Indian testers.

2. What is the Purpose of Weekend Testing?

Weekend Testing is designed as a safe area for testers to learn more about their craft. There is no default testing philosophy associated with the initiative. As stated previously, Weekend testing is a place where testers can learn and practice in a safe environment, have an opportunity to practice their testing craft, learn by interacting with other testers of varying skill levels and experience, and learn about new tools and testing techniques.

Testers can ask questions and can explore ideas openly without fear of ridicule or worrying that their efforts or ideas will be seen as wrong. Each session is planned with open expectations, so participants apply their own experience and use their own skills approaching the mission – which creates rather unique opportunity to see same problem being solved in a variety of ways. From the main Weekend Testing site, the mission of Weekend Testing is to be “a platform for software testers to collaborate, test various kinds of software, foster hope, gain peer recognition, and be of value to the community”. [1]

3. Weekend Testing Chapters

When the original Weekend Testing group was started in Bangalore, the response to the initiative was very positive. So positive, in fact, testers from other areas in India, and in other countries, wanted to participate. The original founders considered the response and decided that having more peer groups in other places and countries was a good thing. With this, the idea of a Weekend Testing Chapter came into being [2]. The largest concentration of chapters at this time is in India. Currently, chapters have been established in Mumbai, Chennai, and Hyderabad, along with the original group in Bangalore. Australia and New Zealand have their own chapter. Europe has a chapter, and the United Kingdom hosts sessions on weeknights, and is called, appropriately enough, "Weeknight Testing". In November of 2010 the Americas chapter of Weekend Testing was founded (serving everyone from Canada to Chile and all countries in between).

Chapters are loose collaborations, and are best considered to be peer groups of testers. While they are typically named after a city or country where founding members are located, there is a strong sense of inter-chapter participation. It is not uncommon to see testers from different continents participating in each other's sessions, and some testers attend sessions no matter where they are or when they take place. Weekend Testing encourages the development of chapters, and wants to see them flourish in more areas.

A peer group of testers coming together and practice testing techniques does not necessarily need to have a formal structure. Nevertheless, the Weekend Testing model has proven to be successful, and as such, if chapters want to be associated with Weekend Testing officially, they agree to follow certain protocols in how they set up testing sessions, as well as conduct the sessions and report the results. Each chapter is free to take on their own testing initiatives and hold discussions that are of interest to their participants. While the process and methods could be used by anyone willing to get together and hold a testing session, it's the posting of experience reports and transcripts, along with formal announcements and the recognition of doing so that sets Weekend Testing gatherings apart from other informal events.

One of the most important reasons for chapters is the time difference between testers. Each chapter holds their sessions at a time that is generally convenient for the participants in their geographical area. Having said that, there is no restriction for participation; many testers actively engage in any session they want to. Time differences may make certain sessions impractical (living as I do in California and in the Pacific Time Zone, if I wanted to participate in a session that was being held in India at 4:00 p.m. IST, I would need to be awake and ready at 3:30 a.m. Pacific Time. Chapters on other continents allow for a broader range of session times for all of its participants. The Americas chapter frequently gets participants from India, Europe and the United Kingdom when we hold our sessions.

4. Facilitating a Weekend Testing Session

The key role in a Weekend Testing session is the facilitator. The facilitator is responsible for many areas that help make the testing mission run smoothly for the participants. The facilitator:

- Determines when a session will be held.
- Sets up the announcement of the session on the main Weekend Testing site.
- Announces to other testers as to when sessions will be held.
- Chooses the topic of the session, as well as the application that will be tested (often by consulting other members of the chapter or other facilitators from other chapters to get ideas).
- Announces the testing charter and mission, which set the stage for the testing and the discussions that follow.

As an example, the Americas chapter tested an application for our second session (WTA-02 – Show Me the Money!" [3]) called DSBudget. Considering the timeframe and specifics of the application, we structured the mission as following:

- Learn and test most common user workflows.
- Test the application and see how it reacts to creating and tracking a budget.

The idea for this early session was to create a simple mission and goal, and to focus attention specifically on that area. While it would be possible to range far and wide with any application, the mission was to see what the most likely workflow would present to the testers, and what issues they would find and report. Because we have testers who participate in multiple Weekend Testing sessions in multiple chapters, it would quickly become stale if we all repeated the same applications and testing initiatives. Besides that, readers of the chat session and the experience report would be able to come up to speed quickly with the ideas and the concepts that were presented in that particular session.

In addition to announcing the sessions and choosing the topics, the facilitators are also the moderators of the session. They set the agenda, they set the pace of the discussions with the input of the other participants, and often may choose to scrap an idea they had for a session in favor of another charter and session based on the attendees and the ideas they provide. The final role of the facilitator is to compile the results and publish them in an experience report and as a chat transcript. They then announce the availability of the results. While each Weekend Testing session is a commitment of two hours worth of time for the participants, a facilitator averages around six total hours from initial idea to publishing of the experience report.

This may sound daunting, but the truth is, anyone can be a facilitator if they are willing to put in the time and guide other testers in participating in these events. The tools necessary to run these sessions are basic, and for the most part, free. Sessions are currently established and managed using Skype. During Weekend Testing sessions, only the IM or "chat" function is used for the full group, unless the purpose of the session is to test Skype functionality such as voice or video. Since chapters have participants from different parts of the world and with different levels of Internet bandwidth, we do not want to exclude anyone from participating. Additionally, limiting the interaction of the main session to Skype's chat/IM tool has the benefit of easily providing a single transcript of the session.

The facilitator sends a message out to participants, announcing that a session is taking place, along with the date and time of the session. Along with the announcement, instructions on how to join a session are sent as well. These are usually done in the following manner:

1. The tester is asked to add the SkypeID of the Weekend Testing facilitator (usually named after the chapter for ease of lookup and remembering; the Americas chapter of Weekend Testing uses the Skype ID "**weekendtestersamericas**").

2. We ask that testers who want to participate in Weekend Testing sessions send an email message or contact us via Skype and let us know that they are going to participate in the upcoming session. On the day of the session, the tester is expected to send a message to the facilitator and say that they want to participate. If they have already sent that message earlier, then the facilitator looks for them and confirms if they will participate. Once the facilitator receives word that they want to participate, they are added to the test group.

3. Most of the sessions are done with web applications. This is because there is a disparity around the world as to how fast applications can be downloaded and installed. What takes for some areas just seconds to download can take upwards of an hour in other places. The other reason for this is that it is common to issue a charter and mission for a session right as the session starts (see #5 below). The reason for this is to not give any advantage to some testers. The goal is to have everyone start from the same point, with the same information. In the event that an application needs to be downloaded and installed, however, then part of the session announcement includes the application to be downloaded and where to get it, so as to allow all participants the ability to start the session with the application(s) needed for the session.

4. During the session, the facilitator is expected to maintain the flow of the conversation. Each session starts with introductions from all testers. The facilitator then announces the charter and

the mission for that particular session. The expectations and the scope of the testing sessions are announced to the group.

5. The testers are then let loose for an hour to explore the application or the challenge presented in the current session. This is where the scope of the testing effort is explained, along with the purpose for the session. This could be a session to evaluate a tool, test a company site or install and work with an existing application. Each session additionally tries to focus on a learning initiative. Some sessions have dealt with heuristics, some with regression testing and others with user experience. In all cases, the goal of the facilitator is to guide the discussion and keep the participants on track. While segues and tangents happen from time to time, the facilitator needs to keep the discussion moving forward. Facilitators typically don't have time to test along with the participants during the session, which is balanced by testing they have performed while designing the mission. Facilitators act as an all-round coaching and support during the session.

6. After the first hour is finished, the facilitator then brings the testers back to the main discussion and initiates debriefing - feedback on ideas of the session, lessons learned, and potential traps and issues that may have become known during the session. This is an opportunity for the testers to speak their minds about things they felt went well and things they wish had been handled differently. Often, the frustrations voiced in one session become lessons for future sessions, so that the opportunity to learn and grow continues.

Testing sessions have had as few as five participants and some sessions have had upwards of two dozen participants. Bigger groups require having more facilitators to run smoothly; typically, one facilitator per ten participants is a good ratio.

5. Benefits of Weekend Testing

There are numerous benefits associated with participating in and facilitating Weekend Testing sessions. These include:

- Allowing for testers of all experience levels to get together and share ideas.
- Provide numerous ways to approach a testing challenge.
- The participants in a session will have the benefit of discussing and considering their approaches.
- Work through different scenarios, to learn from each other and have a concrete and documented source in which they can build credibility and a reputation.
- Sessions are brief and focused.

While it is helpful to have had experience in certain test areas, it is not required most of the time. Both novice testers and experienced testers can participate in sessions, and indeed the most valuable learning and most memorable sessions are those where new testers and veteran testers work together. Pair testing is encouraged, and the expectation is that testers who are veterans will be willing to pair with novices wherever possible. By doing so, new testers get the chance to benefit from the experiences of veterans, and veterans often get an opportunity to have their own methods and approaches examined in a new light and context by newer, less experienced testers. The learning goes both ways.

Many times, sessions have guest facilitators who schedule time to participate with the weekend testing groups. In the past few months Dr. Cem Kaner, Michael Bolton, and Jon Bach have both volunteered to facilitate sessions and bring topics for the groups to discuss. It's a great opportunity to have world renowned experts participate in these events and do so free of charge and have the opportunity to learn from them in this capacity.

Many times, a fearless tester offers to bring in their own application they are working on and bring up a testing mission based on the application they are currently working on in their day job. Provided no

intellectual property laws are being violated, this is a perfectly acceptable and often desired benefit of Weekend testing from many companies' perspectives.

Each testing session is recorded. The chat transcript is uploaded to the main Weekend Testing web site (http://weekendtesting.com) along with experience reports written by and about the participants. Since each tester that participates has the option of having their name listed with the experience report and inside of the chat transcript, this creates a simple way for a tester to highlight and demonstrate competence in a particular testing area. Simply opening a search engine and typing in "[Tester's Name] Weekend Testing" it becomes simple to verify that testers are actively participating in sessions. Additionally, it is possible to follow along with experience reports and chat transcripts to see what various testers are learning and teaching each other, and what finds they have made while participating in Weekend Testing sessions.

In addition to the experience reports and chat transcripts, there is also a permanent repository of test reports and bugs that have been filed. Sessions utilize a hashtag system to help identify areas of interest, as well as to make a simple method to parse lines of the chat files using scripts if desired (see "Section 7, Lessons Learned in Weekend Testing" for examples).

This way, testers can go examine bugs from previous sessions and consider additional test ideas. Also, these chat transcripts are available to all, in many cases long after the test session has been completed. This gives an opportunity for those who are shy or unsure if they would like to participate in sessions themselves a chance to, at their leisure and at their own pace, review previous sessions and examine missions, charters and real time experiences, with no time limit or impatient peers to get in the way of their understanding the questions and issues raised during the session.

6. Challenges of Weekend Testing

One of the largest challenges is associated with one of its key benefits. Since sessions are designed to last for two hours at the top end, testing missions are developed to fit that time frame. There are times where longer and more extended testing sessions would be valuable, but they are not currently part of the process. In addition, test initiatives are standalone in most cases. Since it would be a barrier to participation to have many serial sessions (as well as require an extended commitment from those who would participate in the sessions), the ideal has been to focus on standalone sessions.

Automation is a special challenge, in the sense that many testers need to have a similar background and understanding to effectively approach testing with an automation framework. While there have been introductory sessions conducted with automation in mind, more involved and advanced topics are currently not covered in the Weekend Testing format.

These are areas that are actively being explored by participants and there have been discussions around how to address some of these key issues. Some ideas that have been discussed have been the idea of having "special interest groups" that consider key areas like automation, performance, load testing, testing education, etc. and those who sign on to these special interest groups commit to possibly more active sessions and sessions of longer duration. Currently, there is an initiative underway with Weekend Testing Americas called "Project Sherwood" that is meant to help address these areas. This initiative is being designed to allow for longer missions (many hours or many days if needed) and with the ability to link multiple testing sessions together under a single framework.

7. Lessons Learned from Weekend Testing

Over the past few years, many different chapters have conducted over a hundred sessions and compiled a corresponding number of experience reports. There are many lessons that we have learned from these

experience reports. To go through all of them would be far to expansive for a single paper, but the following are some examples of sessions we help and lessons that we learned. Please note that the sessions listed are not in chronological order and are a mix of sessions from various chapters, not just the Americas chapter.

Lesson 1: Geography Matters
Session: WTA-01: Let's Dance [4]

During the first Weekend Testing session that the Americas chapter hosted, we tested an open source version of the popular Dance Dance Revolution dance game called Stepmania. This was my first full facilitation of a session, and I was surprised to see that there were not just testers from the Western hemisphere, but from Europe and from India as well. The supporting files were easily downloaded by many of the American testers in a matter of minutes. Some of the testers in India, however, were not able to download the application at all during the course of the session. Several of the testers were still downloading the application after the session had concluded.

The lesson learned from this was that different geography areas have different levels of Internet capacity, and if there is a requirement to download an application, that requirement must be announced well in advance of the session, so that all testers have the ability to start the session on the same footing. While this may give some testers a "head start" that is outweighed by the need to make sure that no tester is left unable to test for the session.

Lesson 2: The Value of Session Based Testing
Session: WTA-08: Quoth the Developer, Nevermore![5][6]

Weekend Testing participant and developer Ben Simo offered up his Software Developer Quote Generator at http://ribbit.cc/ for this session. Ben participated in this session as the stakeholder. The primary purpose was to explore the power of session based test management and methods that could help us maximize the potential of the test group in a short period of time.
The goal of the attendees was to explore the product and plan testing missions – charters. Each pair or team split off and worked on their own testing charters, effectively splitting the testing mission up into a number of attainable targets. Each group then broke down the areas further and created specific charters in those individual areas. At the end of the session, all charters and missions would be combined into one document. Through this, the session attendees decided to optimize their reporting directly into the session notes by creating hash-tags:

- Bugs used the hash-tag #bug,
- Issues used the hash-tag #issue
- Charters were added with the hash-tag #charter.
- Focus areas within the charter were added with the hash-tag #area.

Not only did this prove to be an effective model for cutting up and creating quick charters and examples for testing, it became a reporting mechanism in and of itself that was quickly adopted by the attendees and used in future sessions as an efficient mechanism for recording in real time their findings.

It also provided a simple syntax to be used with scripts to parse the information and split it out based on the hash-tags used.

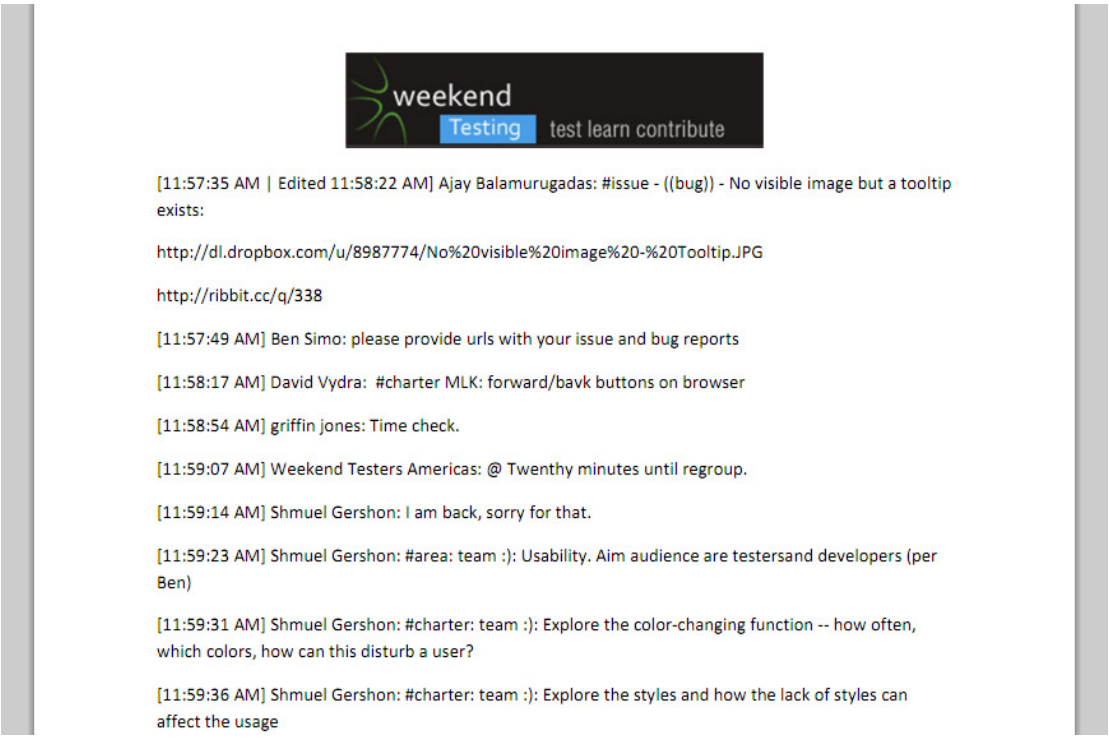


Figure 1: Chat transcript with hash tag examples (http://weekendtesting.com/wp-content/uploads/2011/03/WTA08-ExperienceReport.pdf).

Lesson 3: Let Divide and Conquer Be Your Ally
Session: WTA-04: Put it on the Board [7]

Timothy Coulter developed http://Corkboard.me as a virtual "Sticky Note" cork-board that is accessible online. It gives the user the ability to create, resize, modify, and add text, pictures and other items. Our mission was to "Employ domain testing techniques ("Divide and Conquer The Data") on the sticky notes in cork-board application to find bugs." As a reference, we provided the participants with the following documents:

www.testingeducation.org/a/DTD&C.pdf [8]

http://www.kaner.com/pdfs/GoodTest.pdf [9]

Think of the fact that this application could be accessed by any number of devices and any number of browsers. How could we test all of the variations, even with a large group of gathered testers? Even with an application that is, on the surface, fairly simple, the potential number of inputs and variables that could affect testing are almost infinite. As a way to bring order to the potential chaos, the attendees split up into small groups and each took a chunk of the testing puzzle. Groups are generally self forming, but if individuals do not coordinate quickly or individual do not speak up, which was the case this time, the facilitator can make suggestions to encourage participants to create pairs, or invite the less vocal participants to pair with the facilitator(s). Each group would discuss their findings at the end of the testing period or at times initiate chats with other groups. Many times, the facilitator needs to poke each group and remind them to share their findings with the main group from time to time. By using these techniques, each of the smaller pairs or teams were able to effectively "Divide and Conquer" the application.

By divvying up the testing into slices of functionality and farming it out to other groups, the potential for effective coverage goes way up, and while there may be overlapping areas, the coverage provided by this group approach and through “Divide and Conquer” tactics can make a big difference in total coverage of an application. While not complete coverage, it gets the group closer to that goal.

Lesson 4: Make Sure the Testers Understand the Mission
Session WTA-03: Rapid Reporter, Mapping the Testing Tool [10][11]

For this session we focused on a tool called Rapid Reporter. This application is a tool developed by frequent Weekend Tester Shmuel Gershon. In additional to being a regular Weekend Testing participant, he presents in conferences, advocates for Exploratory Testing, and he developed Rapid Reporter for the purpose of supporting exploratory note taking. The focus of the session was intended to be looking at Rapid Reporter and learning about the application with the goal of being able to use the application for future testing sessions. In fact, the stated mission was “Produce either a list or a mind map of the features of Rapid Reporter *with the goal of guiding a future session of Weekend Testing*“.

The session itself yielded a treasure trove of information about the application itself, including great suggestions for modifying and improving the tool. One of the first takeaways from this session, outside of the great features and desired improvements for the Rapid Reporter application, was the fact that there was some confusion as to what the actual testing session was supposed to accomplish. While traps and issues were being discussed we realized an unintended trap was there. The mission: “Produce either a list or a mind map of the features of Rapid Reporter with the goal of guiding a future session of testing” could be interpreted two different ways:

1. Understand the features of Rapid Reporter so that we could test Rapid Reporter more thoroughly (i.e. directly testing Rapid Reporter).

2. Understand the features of Rapid Reporter so that we could test another application and utilize Rapid Reporter in the process.

Very often, testers receive a charter or a mission, and think they understand it, and then go off to test with the information they received and the way that they understood it. This session showed us that it is important to make sure that all testers understand clearly the intention of the mission, and that asking questions to help clarify the mission is critical to the success of the session. Moreover, this is an important example that testing without exploring about client’s need produces irrelevant results and brings little to no value.

Lesson 5: The Challenges of Understanding Cultural Norms
Session: EWT-39: Let’s Date [12]

This session was based on the real world experiences of Markus Gaertner and was shared with the European Weekend Testing chapter. Imagine a large order is hinging on your ability to solve a particular challenge. In this case, a customer in Nepal is interested in your company's application. All looks good, but they have one important request. The application must support Nepali dates in combination to the standard Gregorian calendar.

Seems like a simple issue to resolve, right? Just map the two calendars and all will be fine. Except for the fact that, for many testers, they don't know going in what the Nepalese calendar is and how the dates would correspond to the Gregorian calendar. How could we implement such a feature? Could we even map these two calendars reliably? Through research it was discovered that the Nepalese calendar is

based off of the Bikram Samwat Calendar, which is decided by a committee every decade. Therefore, dates twenty or 30 years in the future are impossible to guarantee.

Through the discussion, a consensus was reached that the requirements would have to be clearly discussed. The limitation about the inability to guarantee dates more than a decade into the future was a major potential issue. Without some study of the past and present implementation of the Bikram Samwat calendar, and the realization that there was no discernable repeatable pattern to the dates, the requirements of any tests would be based on erroneous information and a promise of quality might be made that could not be delivered upon. For a number of the testers, the idea of a calendar system that changed that frequently was both disconcerting and frustrating. The key take-away was that researching the calendar and the system for assigning dates provided critical information that shaped the requirements and the potential support of the date matching rules. Additionally, it reminded the testers about the value of "testing the requirements". One could question if the requirements as designed were adequate for this system. What happens if a date must be calculated that falls outside of the expected parameters? Think of a 30-year mortgage program; it would require a specific date to be effective. Not being able to calculate those dates could prove very costly in the long run.

Lesson 6: Usability Testing and the Value of the "Persona"
Session: WTA-12: What Are You Watching? [13]

This session focused on the SideReel.com website. The goal was to have a look at the aspects that we commonly associate with "user experience". Even with testers it is possible to get different interpretations of what feels right in an application and what doesn't. Having multiple testers focusing on user experience can open up discussion points for an application even when the testers are not the target users for the application.

The mission for this session was to follow the basic flow of the application (as seen by the developers and the company). With this basis, the group was let loose for an hour and asked to put themselves into the shoes of a "typical user", and with a broad idea as to what a typical user might look for and find appealing in the site, or find less than appealing.

There were many areas that were questioned and evaluated, specifically with the ideas of what would be considered a good approach to searching for show content, dealing with the links that were found, variance in display on various browsers, and again, really trying to get into the mind of who might be the primary user of the SideReel product, and numerous suggestions as to how to make the experience better for the user. One of the most powerful tools in this process is the idea of the "Persona" or trying on the roles of different users. A middle aged man is going to approach the SideReel service differently than a teen-aged girl would, not just in the choices of the shows they watch, but with their expectations as to how easy it is to find links, and what threshold for experimentation each would allow. Playing these different personas will help greatly with getting a better feel for the application and the best way to interact with it.

Lesson 7: Clear Communication of Issues and the Value of a Close Read
Session: WTANZ-11: The Critical Thinking Skill of Close Reading [14]

As testers, we re all expected to communicate clearly and effectively, but what can we do when those who have come before us, or developers who we are working with, do not follow the same rules?

The Australia/New Zealand chapter of Weekend Testing hosted this session. The goal was to help testers recognize the value of a close reading and applying critical thinking skills to the process. Imagine that you have two bugs, and they have been merged together as duplicates. The question that came up was to see which of the two bugs should have been used and which was marked as a duplicate and discarded? Was it the one selected as the representative case the appropriate one to keep? Should a third bug have been written that more effectively combines description of the other two bugs?

The session focused on taking several bugs marked as duplicates and comparing/contrasting to see which was the most effective at communicating the issue in question. The takeaway that I took from this session was that, often, we gloss over the things that we read, or we read into it what we want to read, because we have conditioned ourselves to focus on particular areas. Because of that, we often neglect to do a close read of the real issue, and we miss important data points that will help with reproducibility or with finding solutions. With a more critical read, or even reading the same issue at different times, we can get more information on each read through.

8. Conclusion

Weekend Testing has the power of harnessing and utilizing the minds and efforts of passionate testers all over the world. Its strength comes from the different experiences of individuals with different experiences and understanding of how to accomplish key goals in their testing. The potential to learn something new is always there, and oftentimes, veterans learn as much from neophytes as the new testers learn from the long timers. The methods and processes used in Weekend Testing can also provide a jump-start to any organizations testing efforts. By putting into place the paradigms and methods used by Weekend Testing, test teams that know little to nothing of each other, or test teams that have been together for years, can make major strides in developing skills, transferring knowledge and helping train newer testers with greater confidence and have fun in the process.

For those who are curious about this approach, by all means consider joining a session of Weekend Testing some weekend and see it in person and in real time. For those who have participated in these events and enjoy the format, consider becoming facilitators of your own Weekend Testing style group. Who knows, with a little practice and dedication, teams could grow much stronger and learn a tremendous amount from each of its members. If you would like to experience the Weekend Testing method for yourself, keep a look out for announcements in various venues about future sessions. From there, get on Skype, and come join us for a couple of hours of fun and productive testing. What's more, if you have a core group that finds this valuable, consider joining one of the chapters in your area, or even starting one of your own. There are plenty of volunteers who would like nothing more than to see more Weekend Testers join the movement, and would be more than happy to help you join the fold. Whether it be on weekends, at night or in the morning doesn't matter. What matters is that testers help each other get better in their craft, and as far as we in Weekend Testing are concerned, the more the merrier.

References

[1] Soundararajan, P. 2011, *Weekend Testing, About Us.*, http://weekendtesting.com/about-us, retrieved on 6/20/2011.

[2] Tuppad, S., 2011, *Weekend Testing Chapters*, http://weekendtesting.com/chapters, retrieved on 6/20/2011.

[3] Larsen, M. 2010 WTA-02: Show Me The Money, Experience Report, http://weekendtesting.com/archives/1552, retrieved on 06/20/2011.

[4] Larsen, M. 2010, *WTA-01: Let's Dance,* Experience Report, http://weekendtesting.com/archives/1527, retrieved on 06/20/2011.

[5] Larsen, M. 2011, *WTA-08: Quoth the Developer, Nevermore!*, Experience Report, http://weekendtesting.com/archives/1828, retrieved on 06/20/2011.

[6] Gareev, A. 2011 *WTA08: Messing with the Smart Guys, Automation Beyond,* http://automation-beyond.com/2011/03/16/wta08/, retrieved n 06/20/2011.

[7] Larsen, M. 2010, *WTA-04: Put it on the Board*, Experience Report, http://weekendtesting.com/archives/1638, retrieved on 06/20/2011.

[8] Padmanabhan, S., 2001, *Domain Testing: Divide and Conquer*, http://www.testingeducation.org/a/DTD&C.pdf, retrieved on 06/20/2011

[9] Kaner, C., 2003, *What Is a Good Test Case?*, http://www.kaner.com/pdfs/GoodTest.pdf, retrieved on 06/20/2011

[10] Larsen, M., *WTA-03: Rapid Reporter: Mapping the Testing Tool*, Experience Report, http://weekendtesting.com/archives/1583, retrieved 06/20/2011

[11] Gareev., A., *WTA-03: Mission Impossible?*, Automation Beyond, http://automation-beyond.com/2010/12/15/wta03-mission-impossible/, retrieved on 06/20/2011.

[12] Gaertner, M., *EWT-39: Let's Date*, Experience Report, http://weekendtesting.com/archives/1441, retrieved on 06/20/2011

[13] Larsen, M., *WTA-12 : What Are YOU Watching?!*, Experience Report, http://weekendtesting.com/archives/2081, Retrieved on 06/20/2011

[14] Compton, M., *WTANZ-11: The Critical Thinking Skill of Close Reading*, Experience Report, http://weekendtesting.com/archives/1563, retrieved on 06/20/2011

Delivering Relevant Results for Search Queries with Location Specific Intent

Anand Chakravarty
anandmc@microsoft.com
John Guthrie
johngut@microsoft.com

Abstract

With a still growing number of users and an ever-growing volume of information available, the Internet presents many interesting and continuously morphing challenges in the area of Search. In addition to the scale of data to be processed, the results of user search queries are expected to be highly relevant and delivered speedily. Presenting these results to the user in a manner that enables them to decide and act based on the information provided is a primary characteristic of a good Search engine. Measuring the accuracy and relevance of Search results is thus an important area in the Testing of Search engines. Considering the high volume of information to be processed, the extremely diverse nature of query-intent and the growing expectation of high relevance from Search users, testing of Search engines requires the traditional QA characteristic of passion for quality, combined with a high-level of comfort with ambiguity and strong automation skills.

When we consider the area of search queries that have a location specific intent, there are other factors that become important along with the usual technical problems. Most queries that have local intent have a higher degree of immediacy in terms of the user's intent to act on the results returned to their queries. There is thus greater expectation of the results being fresh and accurate. With the growing market for Mobile devices, searches with local-intent are becoming more popular. As a tester, when presented with measuring how well a Search engine performs for such queries, it is important to understand the scale and variety of queries involved. Because there is a high level of ambiguity and variation in search queries and results, statistical metrics are a natural tool to measure Search engine quality.

In this paper we cover the methods used to obtain metrics for measuring relevance of search queries with local intent. Testing is done while fundamental components are in a dynamic state: rankers, intent detectors, content, location identifiers, etc. A QA team that comes up with good metrics to measure the quality of search results in such scenarios increases the quality of Search Experience delivered to their users, and helps to evolve the quality of solutions implemented in this extremely challenging problem space.

Biography

Anand Chakravarty *is a Software Design Engineer in Test at Microsoft Corporation. He has been testing online services for most of the past decade, and is currently on the Bing Local Search Team.*

As a serial entrepreneur, ***John Guthrie*** *founded, sold, and/or participated in the IPO of companies in various fields. He has led teams to develop, test, and patent technology for Digital Rights Management, Data Storage, and Web Site Modifications. Now at Microsoft, he is working on ensuring the quality of the results for local-intent queries at Bing.*

1. Introduction

User-perceived relevance of web-based Search results is a key metric that defines the success of a Search engine. It is therefore vitally important that the metric be correctly and clearly defined, accurately and systematically measured and, finally, evaluated and improved over time.
To understand what is meant by user-perceived relevance of a Search result, let us consider, as an example, the Search query "Pacific Northwest software quality". The first question to answer here is: what is the user's intent when they search for "Pacific Northwest software quality"? Without a lot of background, it could be assumed that the user might be looking for:

- an organization/corporation by that name,
- or a conference by that name,
- or book with that title,
- or a study explaining why software quality in the pacific northwest is lower/higher,
- or a recent news item relating to software quality in the region, and so on.

Once the Search engine has arrived at some judgment as to user-intent, it then searches its web-index (a database storing the Search engine's snapshot of the Internet) for web-pages that are relevant to that intent. Let us say that, in this case, the user-intent was about a software quality conference or organization in the Pacific Northwest. And let us say the Search engine identified that user-intent correctly. Based off that intent, the Search engine looked up pages in its web-index and then returned the results showed in Figure 1:
The next question to answer is: how relevant are the results that are returned for the query. Let us consider the results shown in Figure 1. The first returned result for this query, which is a link to www.pnsqc.org, is highly relevant. The 2nd and 3rd results, which are links to previous conferences, are also relevant, although less so than result number 1. The 4th returned result, which is a link to an Art Design web-site, is irrelevant to the assumed user-intent. While the returned page does contain the terms Pacific Northwest and Quality, it is not related to software. The 5th result is also relevant, although it is another organization making a reference to an older PNSQC conference.
From a user's perspective then, result number 1 is the most relevant. Results number 2 and 3 could be improved by replacing them with more recent conferences. Result number 4 should have been ranked lower. Result number 5 should have replaced result number 4, and other more recent conferences should probably have been returned and ranked higher than result number 5 as well.
In terms of rating the results then, we could rate result number 1 as Perfect, results number 2 and 3 as Good, result number 4 as Bad and result number 5 as Good.
There are thus two features in measuring the relevance of results returned for a particular query: the degree to which a result matches the user's intent, and the position (rank) of a result in the set of returned results. A highly relevant result returned at a lower position is clearly less useful to the user, same as a less relevant result being returned with a high rank.
The parameters impacting the user-perceived relevance metric that have been defined so far are fairly static. The dynamic nature of the Web, and of the reality captured on the Web, add another level of complexity to accurately defining and measuring user-perceived relevance. Let us consider another example to understand that, in this case the query "U.S. Open results". Depending on the time of the year this query is performed, the user-intent might be to see results for the U.S. Open Golf Championship, or the U.S. Open Tennis tournament. A Search engine that returns different results for that query depending on which event is closer to the time the query is performed would be returning more relevant results to a user.

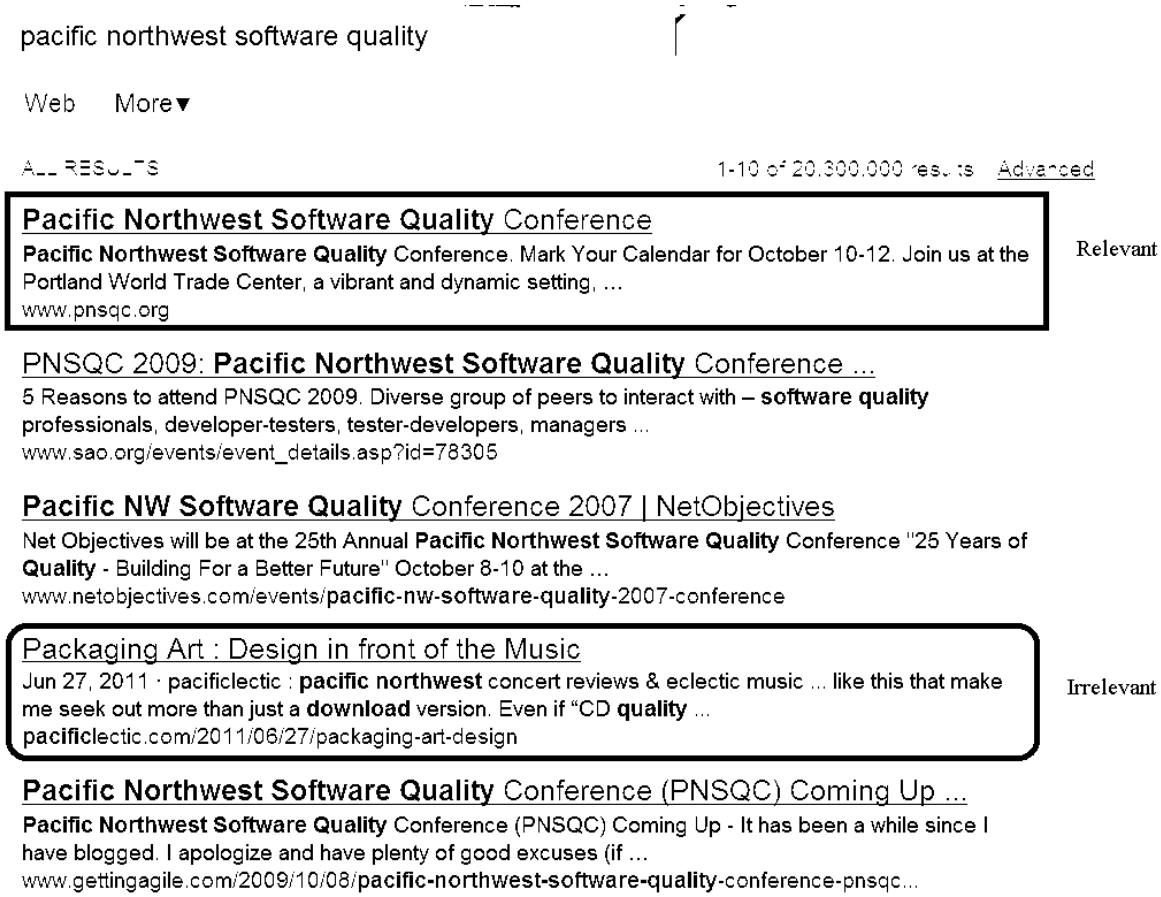


Figure 1

Quantifying the metric of user-perceived relevance involves rating of results for Search queries. Clearly, the number of unique Search queries that a Search Engine would receive is a very large population and it is computationally infeasible to enumerate all queries. For testing and measuring relevance, we therefore use a sample of all queries occurring in the real-world. A well-defined sample would closely match the distribution of queries in the real-world and also account for the frequency with which a query occurs. This means that it would have a mix of popular and rarely used queries, and there would be a weight associated with each of them, the more popular queries getting a higher weight. There are implications of working with only a sub-set of actual queries, because a sub-set is never going to comprehensively match the real world. These shortcomings should be correctly accounted for in calculating relevance metrics. With that in place, the sub-set of queries are then run through the Search engine and the results obtained for those queries should then be rated by human judges. These ratings could involve assigning a number to each returned result, such as 2 for Perfect, 1 for Good, and 0 for Bad. The guidelines for rating should be clear, logical and easily understood. . Also, because the top results returned for a query are more important than results returned at lower positions, only a limited number, say 5 or 10, of the top results for a query should be considered in measuring user-perceived relevance.

Once we have the metric thus quantified, they then become a key input for driving improvements in the Search Engine. To effectively drive such improvements, the collection, storing and evaluation of these metrics should be done in a well-organized and repeatable manner. The variables affecting the calculation of the metrics should be minimized. The numbers reported should be reusable over multiple iterations and across multiple sets of queries. And the trends of these metrics over time and different versions of the Search Engine should be easily viewable to help understand the impact of different changes and to evaluate the progress made over time.

All of the above are applicable to a wide-range of Search Queries. There is a certain category of queries that however require further tuning of the way their user-perceived relevance is calculated. These are queries that have a Local intent. Consider 2 simple queries: ".NET garbage collection" and "pizza". The relevance of results returned for the first query is independent of the location from which a user performs the query. In contrast, for the second query, the relevance of results returned for it vary according to the location of the user. For a user from Portland searching for "pizza", results located in Seattle would be irrelevant, and vice-versa.

In this paper, we explain in detail some of the methods used by the Bing Local Search team to measure user-perceived relevance of Bing search results for Local intent queries. We start by covering the creation of query-sets to measure user-perceived relevance. Once the query-sets have been created, they are run through the Search Engine and results are obtained and stored. These results are then evaluated by judges. The judges follow a set of guidelines that helps them rate results returned for a query. The guidelines capture the steps that we used above to identify relevant and irrelevant results for the query "Pacific Northwest software quality". Each result gets a score, which could be a number like 1 through 5 or a tag, say Perfect, Good, etc. A combined rating for the results returned for a query and the position at which each is returned are then a measurement of user-perceived relevance for that set of queries for the version of Search Engine that was tested. The approach and methodologies we have used have helped accurately understand the relevance of results returned to users performing queries with Local intent over multiple versions of the Search Engine.

2. Creating Query Sets for Measuring Relevance

The number of unique Search queries performed by Internet users is of an extremely high order, even if the domain is restricted to queries with Local Intent. Analysis and evaluation of Search engines must then, by necessity, work on subsets of the entire corpus of possible queries. For testing the first version of a Search engine, in the absence of real world data, a realistic estimate of expected queries is essential. For subsequent versions, data of queries from real-world usage of the Search engine helps make more accurate evaluation of Search engine relevance. A mechanism to log such queries for this purpose, while respecting user-privacy and maintaining anonymity, is therefore a requirement. Privacy of users is maintained by not storing the source IP-address of the query, and only storing the query text and the query location at city-state level.

From the available set of real-world queries, we then create a set of sampled queries to be used for measuring user-perceived relevance. A random sampling ensures a better reflection of real world distribution and thus will eventually yield more accurate metrics.

In our analyses of Search queries, it was found that distribution of query-frequency fits a Zipfian distribution. At the head of the curve are popular queries that contribute to a high percentage of Search traffic. They are followed by a long tail of unique queries that are searched for with low frequency. A sample distribution is shown in Figure 2:

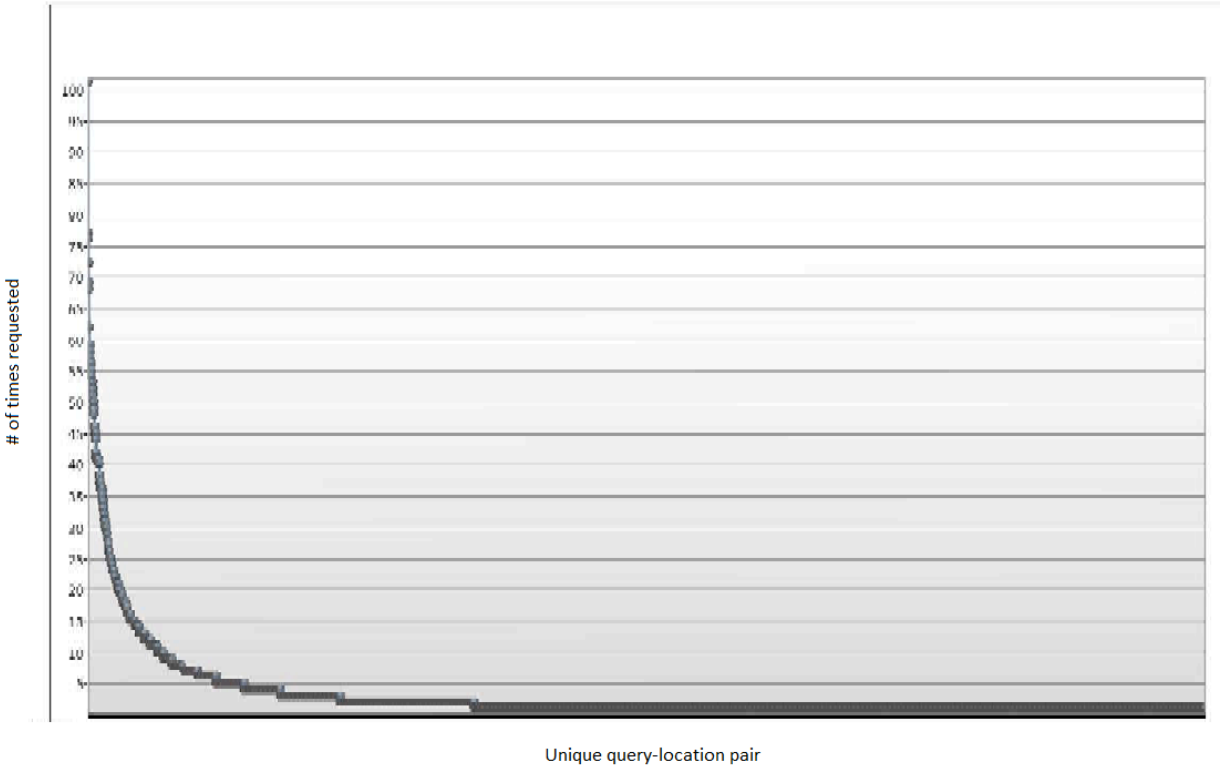


Figure 2

To properly account for this distribution, we apply weights to the sampled queries depending on the frequency with which a particular query occurs. Queries that have a higher frequency thus contribute more to the metrics measuring user-perceived relevance. This approach helps the Search engine be better tuned for higher-frequency queries while continuing to measure relevance for queries that occur with much less frequency.

Queries with local-intent require some additional considerations in creating query-sets in order to correctly to measure the relevance of results returned for such queries. The first consideration is measuring the correctness of the Search Engine in identifying queries that have local intent. One extreme design is for a Search Engine to assume that all queries it receives have local-intent and then try to obtain results for all queries. This obviously is problematic both in terms of returning relevant results for queries that do not really have local-intent, and in terms of the performance implications of using components of the Search Engine that do not have to be exercised. Based on analysis of query-patterns and observed usage, we could arrive at an expected percentage of total traffic that realistically has local-intent. Query-sets created for measuring relevance of local-intent queries should have a mix of queries with local and non-local intent that match the real-world distribution of local and non-local intent queries. One of the metrics calculated should include the percentage of queries from the set of input queries that are triggered as having local-intent.

Another consideration in creating query-sets for measuring relevance of local-intent queries is the actual location of the query. Let us consider an example of such a query, “medical clinics”. Depending on the location from where this query is issued, the user expects different results. An example of a query with non-local intent is “u.s. open winner”. For the latter query, it is safe to ignore the location, because the expected results are independent of the geographical source of the query/user. In creating a query-set to measure relevance of queries that do not have local intent, “u.s. open winner” with a location of “Chicago, IL” is the same as “u.s. open winner” with a location of “Los Angeles, CA”. For measuring relevance of queries that have local-intent however, “medical clinics” from “Chicago, IL” is different from “medical clinics” from “Los Angeles, CA”. Therefore, in a query-set for measuring relevance of queries with non-

local intent, “u.s. open winner” would only occur once, whereas in a query-set measuring relevance of queries that do have local intent, the query “medical clinics” could appear more than once, each time with a different location. If the query-set were to contain “medical clinics” only once, with just one unique location, then the metrics won’t capture the reality of users performing the query from different locations. There is therefore a likelihood that the metrics will either miss improvements made to the Search Engine for such queries, or incorrectly report improvements for such queries based on just one location. This is an important distinction in the methodology of creating query-sets to measure relevance of queries with local intent.

There is another variation in local-intent queries that must be taken into account for creating query sets to measure their relevance. That is the difference between implicit and explicit query-location. In the query “medical clinics”, the location for which results are expected is implicit. By contrast, the query “hotels in Las Vegas” has the location of expected results specified explicitly in the query. So while the latter query does have local intent, the results expected are independent of the location of the user making the query. i.e., a user in Florida would expect the same results for “hotels in Las Vegas” as a user in Iowa. Query-sets for measuring relevance of local-intent queries should have a realistic distribution of queries with both implicit and explicit locations.

In addition to the above broad considerations in creating query-sets for measuring relevance of local-intent queries, there are other features that are useful to take into account depending on the focus and priorities of the Search Engine. A search engine might have different versions depending on the market for which it is designed. A different query set would be required to measure relevance in each of those markets. At the same time, the same search engine might be used to measure relevance of local-intent queries independent of location. For example, the same version of a search engine could be used to answer a query for a user in, say, Boston MA, for local-intent queries with relevant results located in the US, such as {Italian restaurants in Los Angeles} and for queries with relevant results located outside the US, such as {hotels in Melbourne Australia}. For such search engines, the query-set should have a realistic mix of queries across all supported geographical locations.

For a search engine that specializes in queries relating to a particular category of business, such as local queries dealing with a strong restaurant-intent or a strong hotel-intent, it is essential to test with query sets that are predominantly made up of queries matching those intents. This involves identifying not just the local-intent for a query, but also the specific business intent. Classifiers that identify such targeted intents are useful in identifying these queries and help in creating query-sets for measuring relevance of queries matching the targeted intent.

3. Measuring Gain, Precision and Recall

Using the query-sets created according to the considerations outlined above, we are now ready to measure the relevance of a Search Engine for that set of queries. To quantify user-perceived relevance, we calculate numbers for 3 metrics: the discounted cumulative gain (DCG) in relevance for a set of queries, the Precision of results returned for a query and the Recall of results relevant to a query.

Consider a query q, for which a search engine returns 5 results, r_1 through r_5. The cumulative gain of the relevance of returned results for a certain position for this query is the sum of the graded relevance of each result up to this position for the query. To simplify measurements, let us assume a result is graded only as either relevant or irrelevant. i.e., relevance values are binary. For such a query, if only 3 of the 5 returned results are relevant, its cumulative gain at position 5 is then 3.

The discounted cumulative gain (DCG) metric helps to account for the position of relevant results for a query. From a user’s standpoint, search results are more useful if a more relevant result is shown higher than a less relevant result. The DCG metric includes the graded relevance of a result and the position at which it occurs.

Because of the variety and distribution of queries received by a Search Engine, the DCG metric should be normalized so that it may be compared across a wide-range of queries. The normalized DCG, or NDCG, for a set of queries at a particular position is obtained by dividing the DCG of results returned up to that position by the DCG of the ideal set of results at that position for the query. The ideal DCG is obtained by

sorting the results for a query by relevance, with the most relevant result at the top and going down to less relevant results. For perfectly ranked set of results, the normalized DCG is therefore equal to 1.
Precision indicates the relevance of a result; the more relevant a result, the higher its Precision value.
Recall measures the completeness of the results returned for a Search query. A result set for a query is said to be complete if all relevant results for that query are returned in that set. In our example query "Pacific Northwest software quality", a complete set of relevant results could be all web-pages relating to the PNSQC. Let us say there are fifty such pages. If a Search Engine returns only 10 of those pages in its set of results for the above query, then its recall for the query would be calculated as 10/50 = 0.2. A higher recall indicates that more of the relevant results for a Search query were returned.

Given the nature of Search Engines, there is a tendency for Precision and Recall to push against each other such that higher recall comes at the expense of lower precision and vice versa. This happens because as the Search Engine attempts to uncover more results, it also surfaces results whose relevance is questionable. Conversely, if the Search Engine is stricter about what it considers a relevant result, it may miss some results on the fringe. As with most statistical quantities, it is important to have accurate margins of error calculated for each of these metrics, calculated for the system being tested.

Using the well-designed query sets defined in the previous section, calculating the gain, Precision and Recall values for the result obtained for those queries by a Search engine yields a good measurement of the relevance of results for a Search query.

It should be emphasized that like most metrics, the numbers that arise out of their calculations are essentially just an indication of overall performance, they do not tell us a lot until we delve into the details. Let us consider a scenario where for a particular version of the Search engine, when it was tested with a set of queries, there was a drop in the overall Recall numbers. It is the nature of a non-deterministic system that for the same input, it won't always produce the same output. Also, for a different version of such a system, the difference in results for the same input will also be different, in a non-deterministic way, for the individual items in the input. Therefore, in the scenario under consideration, not all queries would experience the same drop in Recall: some queries might even see an increase in Recall with a new version, while most may see a drop. To identify the cause of this drop in Recall, we would first identify queries that experienced a drop in their Recall. Then we look at each of these queries and try to identify the root-cause of the drop. As we analyze more queries, we should place queries into buckets, with each bucket mapping to a particular root-cause. Once we are done with identifying the root-cause for all queries that experienced a drop in Recall, we then look at the number of queries impacted by a particular root-cause. This then drives improvements of the Search engine, as we fix the root-causes depending on the scope of their impact.

In a smoothly running system with a high degree of repeatability, the numbers are expected to be more reliable. In products developed under the Agile model where a lot of components change simultaneously, these metrics tend to show a higher level of variability, and it is then more important to look at trends and apply accurately calculated margins of error to these metrics.

Furthermore, as the Search Engine goes through multiple iterations, it becomes essential to recalculate new baseline values for these metrics, both to measure the impact of each new version and to better understand trends in Search Engine quality over long time-spans. Additionally, as usage patterns change, the set of input query-sets may also require modifications. Such updates should also drive recalculating of baseline values for the metrics.

4. Guidelines for judges

The process of measuring user-perceived relevance ultimately involves human evaluation of results for Search queries. These are performed by a team of human judges. The judges are shown either a query with all the results returned for it or a query-result pair. They rate either the whole set of results or each query-result pair. In order for the ratings to be accurate, a well-defined set of guidelines for rating each result should be provided to the judges. These guidelines would be different for each domain of Search. For example, for the component of a Search engine that is focused on providing results for queries relating to News articles, the guidelines might include a definition of freshness of a result. Similarly, for queries related to Movies, the guidelines could include defining the completeness of information

presented in the results: do the results contain show-times, and ratings? For queries with Local intent, as mentioned earlier, it is important to consider the location for which a query is performed. A high-level overview of guidelines used for rating results for Local Search queries is shown in Table 1. The table shows example queries and a result for each. Using a scale of Perfect, Excellent, Good, Fair and Bad, the table explains to the judges what rating would be appropriate for each example. The judges then use these guidelines to manually rate all queries and corresponding results that are presented to them as part of the relevance measurement process. These ratings are then used in calculating the SNDCG, Precision and Recall metrics for each query of a set of queries. As the Search engine evolves, and data in the real world changes, these guidelines may have to be updated and fine-tuned so that the ratings given by the judges remain accurate and useful for the purposes of calculating relevance metrics.

Query	Local Search result returned	Rating
{Ikea in Seattle, WA}	Ikea at 601 SW 41st Street, Renton, WA	"Perfect"; The nearest location of a specialty store
{dilbeck realty los angeles CA}	Dilbeck Realtors GMAC Rl Est 2486 Huntington Dr, San Marino, CA 91108	"Excellent"; A relevant result close to Los Angeles was returned.
{lane bryant} from Elmhurst,Illinois	Lane Bryant 101 Oakbrook Ctr, Oak Brook, IL 60523	"Good"; Result matched query, although it was not exactly at the specified location.
{ Pets Mart Seattle}	Pets Mart at 1203 N Landing Way, Renton, WA	"Fair"; The Pets Mart franchise has a location in the city of Seattle
{Hong Kong Express Seattle}	Hong Kong Airlines	"Bad"; The result is not relevant to the query. The user is looking for a restaurant and is returned an airline.

Table 1

It is important to be aware of the subjective nature of these ratings. The same query-result pair could get a different rating depending on the human judge who is assigning the ratings. To account for this, we work with a large set of queries, and assign multiple judges to a query-result pair, which helps to smooth out such variations between judges. Such variations also reflect real-world user-perception of relevance of Search results, and are therefore acceptable to a certain extent.

5. Obtaining and analyzing the metrics

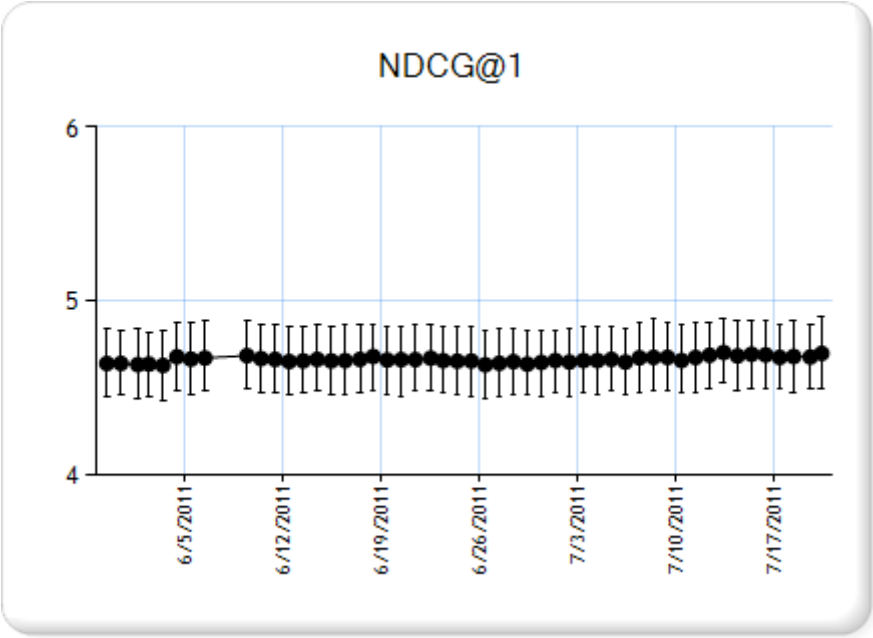
At this point, we have obtained the queries that represent real-world usage, defined the metrics used to measure user-perceived relevance and provided guidelines to human judges to rate results for Local Search queries. The next step is to actually use the queries to obtain results, get them judged and then calculate the metrics. The metrics are calculated for multiple versions of the Search engine.

To obtain results for a set of queries, we create automation tools that run each query through the specified Search Engine version and obtain results for each of them. A database to store results for each execution created, which is useful in evaluating trends over time and also for any subsequent investigations and analyses.

The results are then presented in a visual manner to a group of judges, who have also been trained in the guidelines to use for rating results for a query. The rating for each query-result pair, or a query-results set, is stored in a database. Using the ratings, the NDCG, Precision and Recall for each query is calculated. These metrics are then calculated for the complete set of queries, assigning weights to each query depending on its frequency; queries that are more popular get a higher weight assigned when calculating their metrics. This helps us to account for using only a sub-set of all queries encountered in the real-world.

The process of obtaining results for a set of queries, getting them judged and then calculating the metrics should be automated for easy repeatability. Once that process is working correctly, we may run it

regularly over different versions of the Search engine and see how the metrics are trending over time. We may also run the process against 2 separate versions of the Search engine and compare the metrics for the 2 to evaluate differences and improvements over time. Figure 3 shows the changes in DCG, Precision and Recall over time for a fixed set of queries hitting a Search engine.



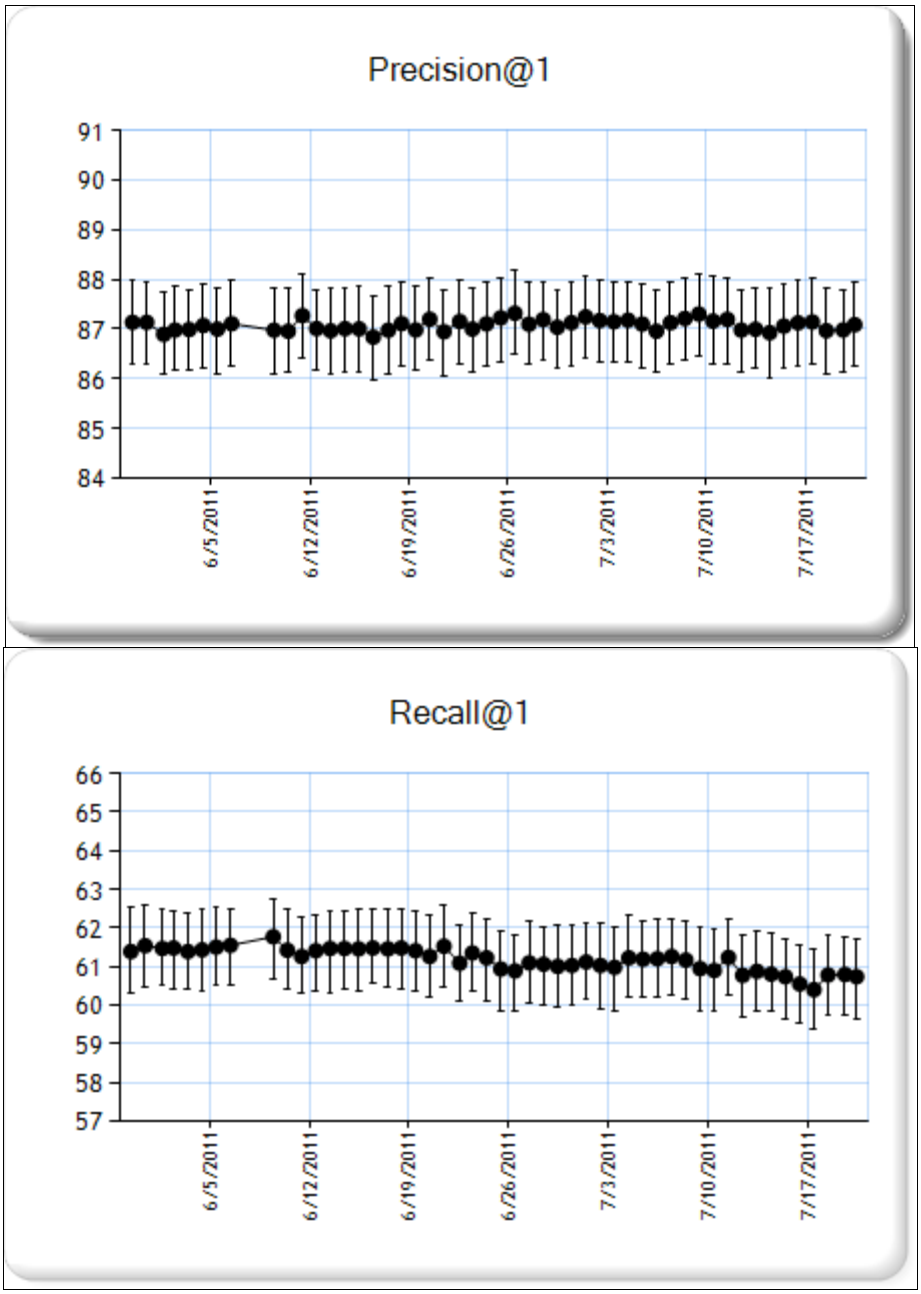


Figure 3

6. Challenges in calculating relevance metrics

As with most Machine Learning systems, there are challenges to quantifying the results of testing a Search engine for location-specific queries. The results returned for the same query changes over time. The calculation of metrics should account for that change. It should also capture the fact that there hasn't been any change when a change is expected. For example, if a restaurant closes, then a Search engine that used to return it among the results for a query looking for restaurants in that area should stop returning it. In terms of relevance metrics, such a change should not affect the Recall value for that query, and if the restaurant continues to be returned, then its Precision value should drop.

Depending on the design, there may also be multiple components that affect the results of location-based Search queries. These could include a spell-checker, a query location identifier, index servers, the data sources and publishing pipeline, etc. Let us consider the query {pizza hug}, in which there is very likely a typo. A spell-checker would usually catch that and return results for {pizza hut}. However, a new version of the spell-checker fails to catch it, or catches it but fails to identify it owing to a misconfiguration or performance issue, then it would impact the relevance metrics calculated for that query. Similarly, changes to the location-identifier might cause Precision and Recall numbers to drop if those changes result in incorrect locations being returned.
All of these moving parts impact the accuracy of relevance metrics. Accounting for them involves dealing with a high-level of ambiguity in the overall system being tested, and correcting the numbers reported so that we have a realistic measurement of user-perceived relevance.

References

Wikipedia, *Precision and recall,* http://en.wikipedia.org/wiki/Precision_and_recall

An Introduction to Customer Focused Test Design

Alan Page

alanpa@microsoft.com

Abstract

Test design, simply put, is what testers do. We design and execute tests to understand the software we're testing, and to identify risks and issues in that software. Good test design comes from skilled testers using a toolbox of test ideas drawn from presentations, articles, books, and hands-on experience with test design. Great testers have great test designs because they have a generous test design toolbox.

One significant drawback of the majority of test design ideas used by many testers is the heavy emphasis on functional testing of the software. While functional testing is a critical aspect of software testing, many test design ideas fail to include high-priority plans for testing areas such as performance, reliability, or security. Test teams frequently delegate these testing areas to *specialists*, and ignore the importance of early testing. Furthermore, when testers manage to design tests for these areas, the testing often occurs late in the product cycle, when it may be too late to fix these types of bugs.

Our team at Microsoft has introduced the concept of Customer Focused Test Design. This test design approach includes an emphasis on testing end-to-end scenarios, real-time customer feedback, future customer trends, and, most importantly, a shift of emphasis away from functional tests, and towards early testing approaches for quality attributes that have the biggest influence on the customer's perception of quality.

This paper discusses the individual pieces of this approach, the success we've had so far, and provides examples of how this change in approach has affected the overall project outcome.

Biography

Alan Page began his career as a tester in 1993. He joined Microsoft in 1995, and is currently a Principal SDET on the Office Lync team. In his career at Microsoft, Alan has worked on various versions of Windows, Internet Explorer, Windows CE, and has functioned as Microsoft's Director of Test Excellence. Alan is a frequent speaker at industry testing and software engineering conferences, a board member of the Seattle Area Software Quality Assurance Group (SASQAG), and occasionally publishes articles on testing and quality in testing and software engineering web sites and magazines. Alan writes about testing on his blog (http://angryweasel.com/blog), was the lead author on How We Test Software at Microsoft (Microsoft Press, 2008), and contributed a chapter on large-scale test automation to Beautiful Testing (O'Reilly Press, 2009).

1. Introduction

Test Design is what testers do – but are we designing and running the best set of tests to ensure that customers desire and crave our products? Our team at Microsoft is exploring a set of alternate approaches to test design with the hope of designing tests that discover issues that affect the customer perception of quality.

Testers design tests, perform them, and make decisions based on those results: Are we done? Do we run more tests? Do we have new test ideas? This is the essence of test design and a large part of what testers do. Lee Copeland's book on test design is one of my favorites on this subject.[1] It's a wonderful book on functional testing, and should be part of every tester's library. However, like many other books on test design, its primary focus is on functional test design, such as equivalence classes and boundary values. Even exploratory testing approaches, depending, of course, on the tester and the testing mission, often drift more towards functional testing than customer-focused testing. Today's test design approaches and techniques are effective, but are they good enough for tomorrow's software? Can we do better?

A good portion of software testing today aims at verifying functionality, and there definitely *is* value in functional testing. Functional attributes are frequently the easiest things to spell out in requirements, and often some of the easier things to verify. Functionality is undoubtedly important, but we may be putting too much of our effort into functional verification while sacrificing the creation of highly usable and desirable software. I believe that testers frequently overemphasize functional testing to the detriment of customer focused testing. No one wants to develop a full-featured and perfectly functioning software program that disappoints or infuriates customers due to issues in performance, reliability, compatibility, or other areas that influence the customer's perception of software quality.

Today, customers demand software that does *more* than just work. Customers want software that is quick, reliable, secure, private, compatible, and usable. Most testers are aware of the importance of these "non-functional" types of testing, but most testing in these areas ends up delegated to the end of the product cycle, to test specialists, or to the customers themselves.

I often hear testers describe themselves as the "voice of the customer." It's critical to remember that while testers can be customer advocates, *we are not the customer*. We are contributing members of the product team and can only be an empathetic stand-in for what customers, providing some insight about how they might want the software to work.

Customer Focused Test Design starts as *testing with the customer in mind*, and contains several aspects that ensure an emphasis on testing for those issues that have the most impact on the customer perception of quality. This paper will discuss four of those aspects: Scenarios, Outside-In Testing, Live Testing, and Shifts & Sparks..

2. Emphasis on Scenarios

One approach to improving customer perceived quality is to build a product around large end-to-end scenarios. These large scenarios are a consolidation of several smaller scenarios or use cases that describe a primary workflow of the product. A project may only have a few of these –usually one or two for each major feature area. Early in the product cycle, scenarios give the whole team a clear idea of the product they're building - and how the product is going to help customers accomplish their tasks.

In their initial form, these scenarios tell a narrative story using the customer's voice. Ideally, they focus on the customer *experience,* not the implementation, and tap into emotion or reveal insight. As the product matures, these scenarios may be adapted to refer to specific product features. An example of a scenario of this type is:

Tina is a **fast-rising** sales manager at a Fortune 500 company. She manages a team of eight people who work at several sites across North America, Europe, and India. In order to manage her team and help them meet and exceed their sales quotas, she **wants** to be able to connect with her team frequently. On most occasions, a short message to her team is adequate, but when context dictates, she often **needs** to speak or see employees who are not co-located with her - not being able to efficiently connect with her team when needed is **frustrating**. She knows that teamwork is critical in order for her whole team to be successful, and she wants to build a culture of sharing and teamwork in order to enable success

We use these scenarios as a guide or charter for focused testing sessions. Because the scenarios are feature agnostic (meaning they describe the task the user wants to accomplish rather than the features they will use), testing tends to focus on ensuring the success of executing the workflow rather than on the testing the functionality of the feature. This also allows for variation in *how* a tester moves through the workflow, giving them a better chance of emulating customer usage patterns.

At this time, our team is at a relatively early stage of the product cycle, but we are already using scenarios as one of the primary progress and status indicators for the state of the product. Low-level metrics like pass rates, code coverage, or bug counts are details that can indicate risk or support a higher-level view of product status. However, we use the scenarios to verify that the product can accomplish customer workflows. It's good to have low-level metrics in case someone needs the details, but scenario status typically gives a much better overview of whether the software will fit the needs of customers. A classic method of reporting scenario status is a standard red-yellow-green report; where green indicates that the scenario is working perfectly, red indicates a scenario fails to complete or is blocked, and yellow indicates some level of intermediate status (see Figure 1). The decisions and details that go into a red-yellow-green report are beyond the scope of this paper, but are worth studying for those interested in methods of reporting project status.

Scenario 1 (green)	**Scenario 2 (yellow)**	**Scenario 3 (red)**
Scenario 4 (yellow)	**Scenario 5 (red)**	**Scenario 6 (green)**

Figure 1 - Example Red - Yellow - Green reporting for Scenarios

These scenarios are a prediction of what customers will do with the software, so the basis of the scenario must come from real customer research and studies and not simply a made up story. You can't predict everything, but basing your scenarios on as much real data as you have will give you a better chance of building the right thing the first time.

3. Outside-In Testing

Many testing reference books such as Jorgensen's "Software Testing"[2] refer to *Levels of Testing* where "testing" starts at unit testing and progresses through functional, system, integration, acceptance testing, and finally into non-functional areas such as performance, reliability, and usability.

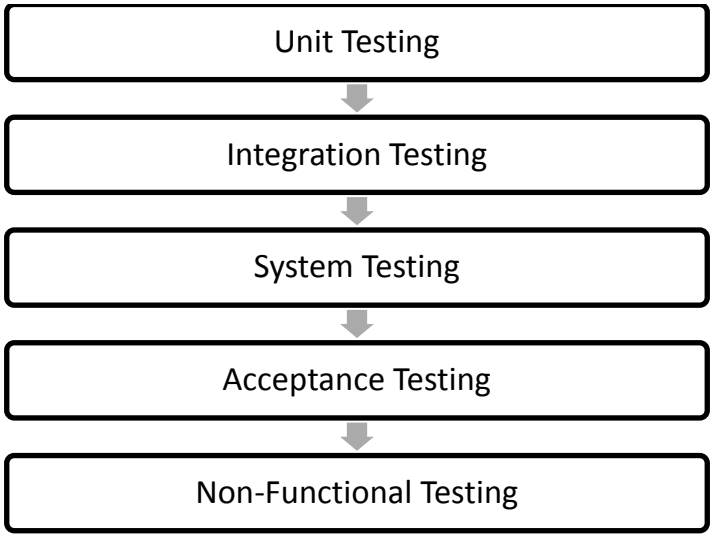


Figure 2 - Software Testing Levels

Typically, testers begin at the level where programmers stop testing. For example, if programmers write unit tests, testers typically start their effort with integration testing. If your programmers do **not** write unit tests, those basic tests may be the beginning of the testing effort.

The testing effort continues through the levels of testing, expanding the scope of testing to scenarios that cover wider areas of the product and towards scenarios with more customer facing attributes. For example, integration testing covers a wider area of the product than unit testing covers, and is closer in what it covers to a customer scenario. The non-functional areas generally cover a variety of scenarios and are a reasonable proxy for how customers will use the software. In this model, the final stage of testing is non-functional testing (e.g. performance testing). Unfortunately, the types of bugs found at this stage are often design issues or other deep-seated issues that nobody can safely address so late in the product cycle.

Most of us have seen or referred to this cost of change curve (Figure 3). While I agree that the cost of finding a bug later in the product cycle is more expensive than finding it earlier, it's important to consider that not all bugs are equal.

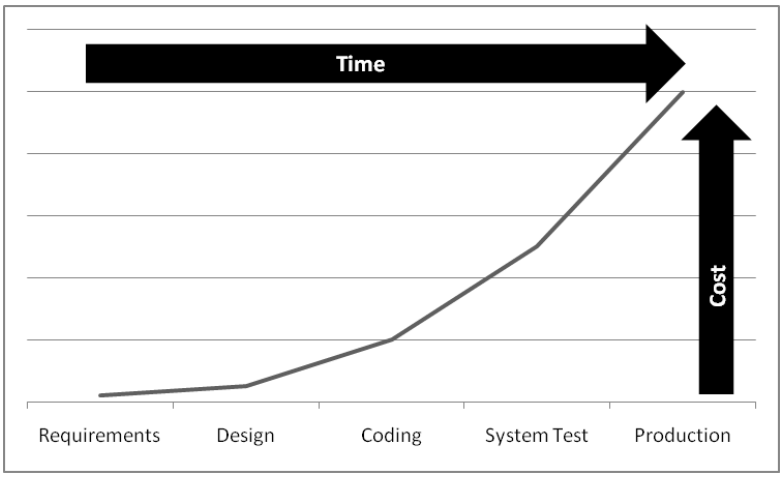


Figure 3 – Traditional cost of change curve[3]

In fact, the cost of change for a performance or reliability bug that occurs due to a design issue could have a much *steeper* curve (Figure 4).

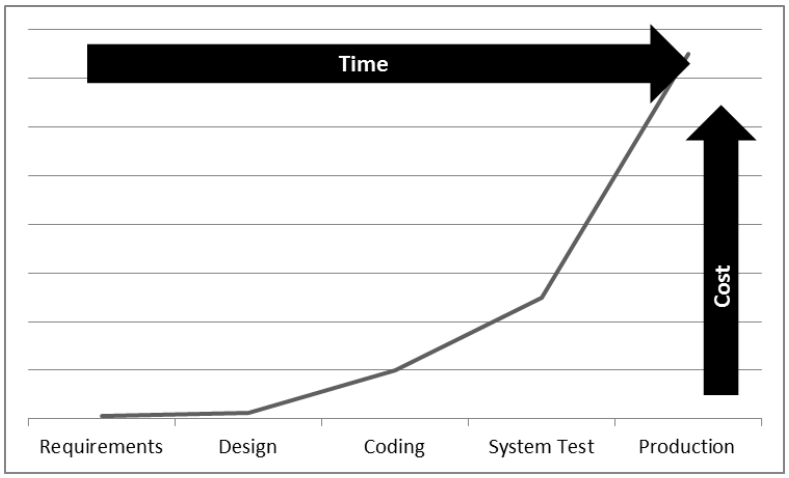


Figure 4 - Cost of change for design issues (hypothetical)

Whereas a functional bug – perhaps an off by one error[4], or a missing hotkey *is only incrementally more expensive to fix late in the product cycle* – often the only increase in cost is the potential extra time it takes to find the offending line of code (Figure 5).

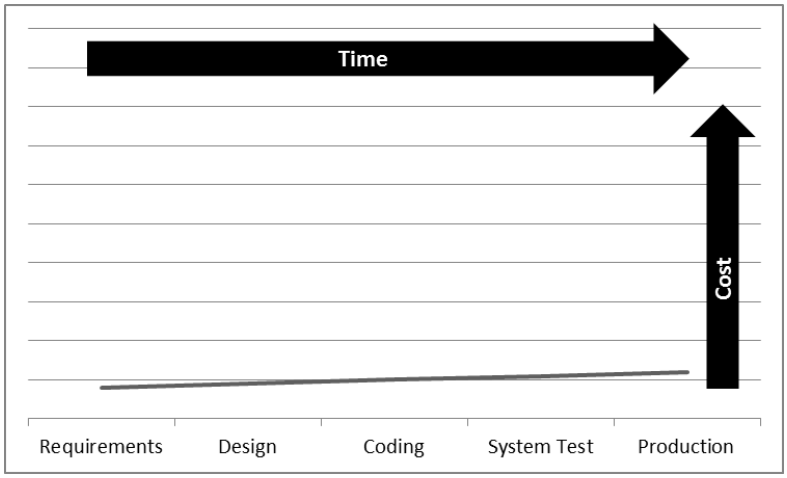


Figure 5 – Cost of change curve for functional bugs (proposed)

The solution we use to address this is an approach I call Outside-In testing (this name comes from a 1973 paper by Allan Scherr where he called the levels of testing "Inside-Out Testing"[5]).

The premise of the Outside-In approach is to put a substantial emphasis on non-functional testing early in the product cycle to uncover as many issues as we can in the areas of performance, reliability, usability, compatibility, security, privacy, and world readiness *before* we verify large parts of product functionality. Instead of progressing through these testing levels from top to bottom, we move to the bottom as soon as possible and work our way backwards (Figure 6).

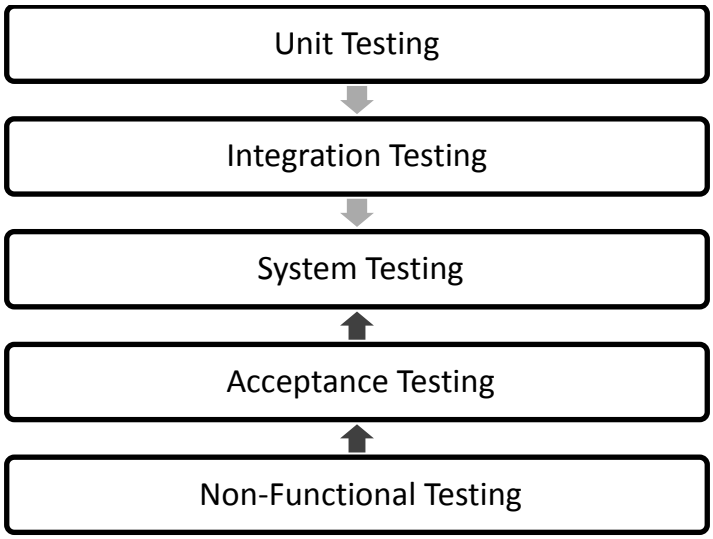


Figure 6 – Outside-In testing

There are two important aspects of this approach. The first is that we do not abandon functionality testing entirely. We've discovered that the product needs a level of functional quality before we can perform non-functional testing, so some early functional verification is necessary. The inverse of this is also true. To perform most non-functional testing, one must test the underlying functionality. For example, if I were to run a "stress test" of Windows Notepad, I may paste a large number of characters into the edit field, save

the file, edit the file, and then repeat the steps. The only thing that adds *stress* to the test is the size of the data. Each of the other steps is a simple functional test.

To ensure that every tester thinks about these non-functional test ideas early in test design, our test design template includes guidelines and requirements for designing non-functional tests at the initial stages of creating tests. Appendix A has an excerpt from this section of our template.

Finally, as with Scenarios, we prefer to report status metrics at a higher level than tests run and bugs found. We look at each of the key non-functional areas and report red / yellow / green status based on the count and severity of the bugs found in those areas.

Table 1 - Example non-functional scorecard

Area	Performance	Reliability	Usability	Security
Status	***Red***	*Yellow*	Green	Green
Bug Numbers	134, 137, 199	111, 174		

.

4. Live Testing

A stream of customer data is critical to react to problems and understand usage patterns. Microsoft collects information on application crashes via Windows Error Reporting[6] and collects considerable amounts of usage data through the Customer Experience Improvement Program.[7] These programs are extremely useful and provide actionable data that we use to improve our products. One drawback of this data is that we generally use the information to improve the *next* version of the product. While we gather this information during public betas, much of it still comes too late to have a significant impact on the current product.

One way of obtaining customer feedback quickly is through Testing in Production (TiP), and the approach is already popular on the teams at Microsoft that build web services, as well as at companies like eBay and Amazon. In the past, web service teams have attempted to build extensive pre-production environments to mimic real-world production sites. Unfortunately, most teams discover it's nearly impossible to replicate production environments and workloads; many issues are uncovered only in the production environment - where the combination of server hardware, scale, and real customer workloads reveal bugs that seldom show up on pre-production systems. The key to a successful TiP approach is extensive monitoring systems that can detect issues that affect customers immediately, resulting in immediate roll back. This enables teams to run tests against production environments and recover nearly instantaneously should an error cause a problem that may result in a customer issue.

We use TiP extensively for server and web components of Microsoft Lync. For example, we run a portion of our dogfood[8] program, as well as a large set of our test automation against production servers that serve real customers. The monitoring and diagnostics systems on our production servers allow us to identify when a problem occurs and roll back or disable tests, as appropriate, to minimize (or eliminate) customer impact.

For desktop client applications, where TiP approaches are not generally applied, we are making a deliberate effort to take some of the best parts of TiP and see how we can apply them to desktop applications. We have the advantage of building a product that the entire company uses for telephony and collaboration, which means that a beta program of twenty to forty *thousand* users is reasonable, even for early adoption.

Additionally, we are attempting to replicate some aspects of TiP with our rich desktop client. Although we can't update these early adopters in real-time, as we can with web services, we strive to gather their

feedback, make adjustments and get them a new build as frequently as possible. Our core team upgrades daily, and we attempt to upgrade many of our dogfood users outside of the core team once or twice weekly. Through customer usage data, we can tell, for example, which parts of the product get the most use as well as which areas get little use. This data prioritizes test investment and contributes to improved risk analysis. For example, if we discover that most Instant Messaging sessions last less than ten minutes and that most messages are between 20-80 characters, we can prioritize functional and non-functional test cases around that data, rather than focus on long strings and extended messaging sessions.

Another example of where we leverage TiP for our client application is A/B[9] testing. A/B testing, where customers see one of two different experiences, is a staple of online services. We can deploy two different versions of Lync and track usage data to help us make decisions about user interface design, usability, and many other customer-facing features. While not directly connected with how we design tests, the test team is responsible for implementing and monitoring A/B testing.

Our goal is to gather customer data and react to that data as quickly as possible, and we are continuing to investigate approaches borrowed from TiP to help us reach this goal.

5. Shifts and Sparks

Our team has pioneered another testing technique that we call Shifts and Sparks. The Shifts come from an initiative done by the Envisioning Lab at Microsoft.[10] The Envisioning Lab, part of Microsoft Office Labs, has performed extensive market research to develop a list of macro forces of change (Shifts) they expect to occur in the social, technological, economic, environmental, and political landscape over the next five to ten years.

For example, one of the shifts is the Content Revolution that anticipates the volume, diversity, and pervasiveness of digital content merging into communication and entertainment. Another shift is Mobile Computing that proposes mobile phones and platforms may surpass PCs to become the dominant personal computing platform.

Our test team uses the Shifts to help *Spark* new test ideas. Testers use a set of cards with Shifts (from the envisioning team) written on one side and Sparks (our test ideas) written on the other. As part of the test design process, testers scan through the Shifts and pick 2-3 that may apply to the area they're testing. Then, they read the Sparks and add tests or test ideas as applicable.

For example, in the area of Mobile Computing, the Sparks include concepts like "Always-on connectivity," and "Bandwidth Dependent Experience." If a tester is testing the Instant Messaging feature of Microsoft Lync, they may read the Spark and ponder the anticipated user experience should network connectivity or bandwidth change during the session. Because of the Spark, they may decide to test on a throttled network, or simulate bandwidth and connection issues to ensure a good customer experience.

Shifts and Sparks are possibly more applicable when reviewing specifications. The team that developed Shifts and Sparks is presenting a poster paper at the PNSQC 2011 conference with additional details on their process and findings.

The goal of Shifts and Sparks is to help testers come up with additional customer scenarios that may not be clear from specifications, code, or implementations. The Sparks help to inspire new test ideas that testers may not have considered using other approaches. These additional scenarios contribute to and enhance the other forms of scenario and end-to-end tests performed by the test team.

6. Next Steps

Our Customer Focused Testing approach is still in a relatively early phase; however, we have already observed several changes that we believe are due to the shift in test design focus. One of the most notable changes is the increase in customer focused thinking across the team. Achieving customer focus was the intent of the effort, but the culture change has been noteworthy. Not only does the team have a customer focus, there is also a "buzz" about relating to the customer. In addition, the non-test disciplines have an increased awareness of customer facing issues due to the increased emphasis on the subject from the test team.

We've received positive feedback from management and peer disciplines on each of the approaches discussed in this paper. Most notably, our program management staff has begun to incorporate shifts and sparks into their planning process. The test team still uses the sparks to generate new test ideas, but our program management team is using the same approach to improve and enhance functional specifications.

One challenge in our approach is appropriately conveying the de-emphasis of functional testing. Because of the focus on scenario-based testing, it has been challenging to reeducate the team to expect some functional bugs to surface in later stages of the test cycle, or to our internal dogfood users. We're slowly making progress with this approach and showing the value in discovering non-functional issues early. In a few cases, we're even seeing developers put more effort into writing unit *and* functional test cases to discover or prevent functional bugs in the first place. One of the original goals of the customer focused test design initiative was to promote developers owning unit and functional testing, leaving testers to focus on end-to-end and non-functional testing. Seeing this change occur, even in a small part of the team, demonstrates the merit of the approach.

Overall, we are pleased with the initial progress but know we have to carry it through to a shipping application before we can truly claim victory. However, we feel that the efforts so far will have a positive impact on post-release customer satisfaction, and believe this could be a useful approach for any test team.

Customer Focused Testing

The following subsections contain the non-functional areas that have the highest impact on the customer perception of quality. Use this section to describe how you will test for these areas.

1. Security

It is important Microsoft customers can trust all our features to be secure. Costs to Microsoft in QFEs and patches, service downtime, credibility, and resulting potential loss of future business are serious.

2. Privacy

Are there tests in place that validate the protection of customer data? How does the software manage the collection, storage, and sharing of data? Are privacy choices properly administered for users, enterprise, parent components? Are all transmissions of sensitive data encrypted to avoid the inadvertent exposure of data? Does the application provide a mechanism to manage its privacy settings during the install, first-run experience, or before data is ever transmitted or processed?

3. Usability

What are the current usability (including accessibility) issues of the feature? What is your approach to identifying new usability issues? What is the usability goal and criteria for this feature?

If the feature is UI (i.e. is seen by an end user), you must include accessibility testing in this test plan. Include testing for high DPI settings, high contrast screen modes, screen orientation, do sounds have visual cues, is everything *sensibly* accessible via the keyboard?

4. Reliability

Discuss your strategy for testing feature reliability. This testing may also need to cover memory/thread leak detection and fault injection depending on the feature area. What are the expectations for reliability when tested under low-resources, unavailable-resources, and high-latency conditions? What is the strategy for long-haul testing of this feature?

5. Performance

Describe how the performance testing will happen. Testing includes simulating load conditions, validating correct operations, establishing the proper mix of operations, and generating the necessary supporting data and environment. Performance doesn't always mean fast – but ensure that performance meets guidelines, standards or expectations.

6. Compatibility / Interoperability

Compatibility is the ability for a program to work with other programs or plug-ins – including previous versions. Interoperability is the ability of the component parts of a system to operate successfully

together. Consider the potential for compatibility or interoperability issues in this feature. For example:

- Will this feature interact with client or server components [from the previous version]?
- Are there any MS or 3rd party add-ins for a previous version of Lync that interact with this feature?
- How does this feature interact across platforms or devices?

Discuss how you will address any issues above. Include a compatibility test matrix if necessary.

7. World Readiness

Discuss the globalization issues with this feature. Consider character issues in text fields as well as formats for date, time, currency, etc.

References

[1] Lee Copeland, *A Practitioner's Guide to Software Test Design*., 2004.
[2] Paul Jorgensen, *Software Testing: A Craftsman's Approach*, 1995; Glenford J. Myers, *The Art of Software Testing*. 1979.
[3] Barry Boehm, *Economics of Software Engineering*, 1981.
[4] An "off by one" error is a functional error that occurs when a program iterates either one too many, or one too few times through a loop. This is common in most programming languages due to zero-based counting and operator mis-matches.
[5] William Hetzell, *Program Test Methods*, 1973.
[6] Microsoft, "Windows Error Reporting," Microsoft, http://msdn.microsoft.com/en-us/library/bb513641.aspx (accessed August, 2011)
[7] Microsoft, "Microsoft Customer Experience Improvement Program ", Microsoft, 2009, http://www.microsoft.com/products/ceip
[8] *Dogfood* is a term used to describe a team that uses its in-progress software to get daily work done. *See* http://en.wikipedia.org/wiki/Eat_one's_own_dog_food
[9] A/B testing presents two (or more) user experiences to customers and tracks data (e.g. number of clicks on a target) for the control and experimental experiences. *See* http://en.wikipedia.org/wiki/A/B_testing
[10] Microsoft, "Envisioning," Microsoft Office Labs, http://www.officelabs.com/Pages/Envisioning.aspx (accessed August, 2011)

Testing Services in Production

Keith Stobie
Keith.Stobie@microsoft.com

Abstract

There are many benefits to be realized by Testing Services in Production when the risks are properly mitigated. Testing in production finds problems at a scale that most groups can't afford to duplicate with a test environment. Using production systems for testing is critical to business success of an effective software service. This paper describes and demonstrates several different approaches to using production systems for testing including: when each approach is appropriate, what prerequisites are needed, and how each approach would be used.

Monitoring of services, Controlled Experiments, and production data are well-known forms of testing. You can also use tracers to follow service flow, do destructive testing (killing services, networks, etc.), and even do load, capacity, performance, and stress testing in production. In short, almost all kinds of testing can be done in production, but how do you mitigate the risk? Testing in production allows customers to benefit (and occasionally to suffer) from the most current advances. Throttling requests or work, exposure control, incremental rollout, and especially superb monitoring are all needed to control production testing risk.

Biography

Keith Stobie is a Principal Software Development Engineer in Test working as a Knowledge Engineer in the Engineering Excellence group at Microsoft. Previously he was Test Architect for Bing Infrastructure where he planned, designed, and reviewed software architecture and tests. Keith worked in the Protocol Engineering Team on Protocol Quality Assurance Process including model-based testing (MBT) to develop test framework, harnessing, and model patterns. With twenty-five years of distributed systems testing experience Keith's interests are in testing methodology, tools technology, and quality process. Keith has a BS in computer science from Cornell University.
ASQ Certified Software Quality Engineer, ASTQB Foundation Level
Member: ACM, IEEE, ASQ.

1. Introduction

Web Services and the cloud are becoming the pervasive way to deliver software functionality to users. As users become dependent on cloud services, they require high assurance about reliability and availability of the service. News reports of failures in public services are frequent (Infosec 2011) (Nguyen 2010). A growing consensus on major web services prescribes testing in production as effective risk mitigation (Ciancutti n.d.) (Hamilton 2007) (Johnson n.d.) (Michelsen n.d.) (Rigor 2011) (Soasta, Inc. 2010). There are still a few holdouts that question testing in production (Rathi 2011).

To make Testing in Production relatively safe, as Hamilton (Hamilton 2007) noted,

> "The following rules must be followed:
> - the production system has to have sufficient redundancy that, in the event of catastrophic new service failure, state can be quickly recovered,
> - data corruption or state-related failures have to be extremely unlikely (functional testing must first be passing),
> - errors must be detected and the engineering team (rather than operations) must be monitoring system health of the code in test, and
> - it must be possible to quickly roll back all changes and this rollback must be tested before going into production."

To keep production metrics correct you must treat data carefully as given in section 1.1. In order to minimize risk while doing production tests you need exposure control as given in section 1.2

Section 2 discusses various approaches to production tests starting from simple Asserts in 2.1 and basic Monitoring in 2.2 to rather traditional testing offline in production environments in 2.3. Controlled experiments as discussed in section 2.4 began as a design comparison technique but can be leveraged by test teams into production testing of new versions, etc. More challenging, after mastering the previous approaches, is capacity testing the live site (2.5), data injection via tracers (2.6) and fault injection (2.7).,

1.1 Isolate Test Data from Production Data

If you are running tests in production and, like most production sites, you are logging for analysis what happens on the live site, you must consider how your tests can potentially perturb the live site data. Some sites may argue that production tests are such a small fraction of their overall workload (say < .01%) that they don't care about the perturbation. Validate if this is really true for the service. Any billing for services can't include tests (except for test billing).

Most services will want to flag and or separate production tests from real production work. Many ways exist to do this. A simple way for HTTP-based work is to include an HTTP Header parameter string, e.g., `Pragma: test`, or an additional parameter string, e.g., `&traffictype=test`.

Then you have to modify your log processing routines to separate and potentially just drop the test data. Failure to remove the test work can have several negative repercussions including:

1) Viewers of the live site metrics may get a false picture (more activity or more errors than are really there)
2) Billing based on live site metrics could be skewed

Some services use a location setting for the browser that does not correspond to any real country or region. Instead of reporting the traffic as coming from a user in India it is reported as coming from a user

in “test.” When they process data to bill their customers, the production tests are automatically filtered out.

Creating user observable test data in production could create a detrimental user experience. Several methods have been used to avoid exposure. Some go for obscurity, e.g., using a language that doesn’t exist. For example, instead of en-US or zh-CN, they might use xx-XX. Other services filter out test data in their front ends. For example, they won’t return test data to any request not coming from a domain inside the production or test environment.

1.2 Exposure Control

Beyond exposure of test data, a variety of approaches is used to control exposure of new versions or test versions. Common practice is to make it available to a limited audience on a limited set of resources and gradually expand both (Eliot 2009). For example, only enough machines and network to run a minimal instance are used initially and then only accessible to a restricted set of people, e.g., employees and then friends and family or external crowd source testers. The initial instance has many names such as slice of production, canary, etc. After a proving period, incremental rollout of the new version is allowed into more production resources in one data center and across many data centers until, when proven, it becomes the default for all data centers. Similarly user exposure may be controlled to Beta users, specific regions, or specific countries.

2. Test in Production Approaches

Test in Production presumes a relatively stable system already verified using standard unit-testing, component testing, and system test techniques. Live sites must be verified for their processes as well including standard operating procedures, incident handling procedures, and escalation procedures. Many organizations and standards cover live site procedures.

Depending on the architecture and maturity of the web service, some approaches are easier to adopt than others. Offline testing in a production environment is decades old and should be easily accomplishable. Asserts are the easiest to adopt and observational monitors are critical to the running of any service. Experiments, A/B testing, are actually a standard marketing approach (MarketingExperiments n.d.) and many third parties (e.g. (Google n.d.)) can help a site do this with no instrumentation changes to the web service except providing the A and B versions. Self-Test is more elaborate, but is a standard coding exercise not unique to services. Synthetic transactions are usually the first test in production approach that requires careful thinking. Tracers are like synthetic transactions but coming from a data perspective instead of a user perspective. Most advanced are Live performance, capacity, and stress tests and also Live fault tolerance tests.

2.1 Asserts

The most basic aspect of Testing in Production is to leave asserts that don’t significantly impact performance in the production code. Services need to have multiple instances of the actual code running both for scaling and availability. You must have no single point of failure in your system (hardware, software, or environment). Your code should be reasonably well tested before going into production, so an assert firing should be a relatively rare event and affect only one or a few instances. Given multiple instances and independent Heisenberg (Gray 1985) asserts firing, asserts are your first line of defense in detecting and recording rare problems via crash dumps or virtual machine images. Through analysis of the asserts fired you can make your system even more robust.

2.1.1 Self-Test

More advanced than simple asserts, is program self-test, just like what is done in hardware. Self-Test generally requires you already have asserts in place and is appropriate for long running services or for background verification of persistent data. Many programs are written with a low priority thread for doing verification or self-test for when the program is not actively used. During these idle times, the verification thread will make various checks such as walk and verify data structures. Similarly many services have low priority background processes that do nothing but constantly look for passive failures (e.g., disk sector becomes unreadable) or data corruption. For example, when you have data replicated three times, you can constantly compare the replicas and, upon disagreement, generally choose by vote the most common two as correct, and repair the third.

2.2 Monitoring

A critical aspect to running a web service is monitoring. Without monitoring, a service is being run blind and it is dangerous to even launch such a service. Monitors which only observe the system are effectively Test Oracles indicating if the service is successful or failing. Monitors measure system and service behavior and raise appropriate alerts or alarms depending on the measurements. A prerequisite or co-requisite for monitoring is an effective alerting and alarming infrastructure. Monitoring requires system capacity that must be planned. Efficiency of the monitors and whether they can be tuned for more detail or less detail will also affect the required resources needed. Observational monitors can be simple measurements like CPU usage above a threshold, e.g., 80%, or memory usage above a threshold. They can be time averages such as average response time over the last five minutes being less than one second. The response time measurement could be based on real user requests and responses. Monitors can be multi-level, as shown in Table 1 where ninety percent of requests must have response time less than two hundred milliseconds.

Table 1. Response Time Threshold observing monitor

% of requests	Threshold
50%	100ms
90%	200ms
99%	300ms

For monitors to be effective, testing of services in at least some non-live environments should be set identical to the production environment to indicate if the thresholds fire too often (false positives) or not enough (false negatives). The frequencies with which measurements are made also affect the performance and responsiveness of the system.

An observational monitor could measure availability by observing how often an error or no response is provided for a request.

2.2.1 Synthetic transactions

A synthetic transaction is a monitor which actively provides input (test data). It may also actively verify the response or rely on existing observational monitors as the Test Oracle. The simplest example is a ping of network system to see if it responds at all. Sometimes availability measures are made based on synthetic transaction pings, but these can be unreliable indicators of actual availability such as you might get from an availability observational monitor.

Almost any functional test can be constructed into a synthetic transaction monitor. Deciding on which tests to run and how often are critical factors in the success of using them to prevent major service issues from arising or to detect issues quickly. Unlike functional tests, many times the test can't rely on a

specific state of the service while it is Live. Thus synthetic monitors may need to use the Delta Assertion pattern (Meszaros 2007).

Some synthetic transactions should always be in use to verify basic service operation. The larger the consequence of failure of a particular test, the more frequent it should be run. A good reporting infrastructure makes synthetic transaction results more valuable. Trends can be observed such as whether some tests only fail at a particular time or under a particular load.

2.3 Offline Testing in Production Environment

For decades services had the luxury of not being available all the time. It was acceptable to have schedule unavailability for some services, for example stock markets. During these offline times, the production environment could be used to conduct tests, but these were not while the system was live.

Similarly, today many services are either provisioned for a maximum load that rarely occurs (e.g., Tax filing day or Black Monday - the biggest online selling day of the year, etc.) or use cloud services that allow dynamic capacity to be purchased on demand. For maximally provisioned services, it is usually possible to schedule taking some of the resources offline during known low usage periods and use the production resources offline for testing. Similarly for dynamically purchased capacity, you can purchase production capacity for testing as easily as for live usage.

Testing offline in a production environment helps by having an environment as nearly identical to the live environment as possible, but still it isn't the live environment.

2.4 Controlled Experiments – A/B Testing

A/B testing is seeing how users interact with two different implementations and, based on some measure of goodness, selecting the better one. For A/B tests, the characteristic is to have production code of equivalent quality so the focus on the experiment is on distinguishing user preference for the designed differences between the implementations.

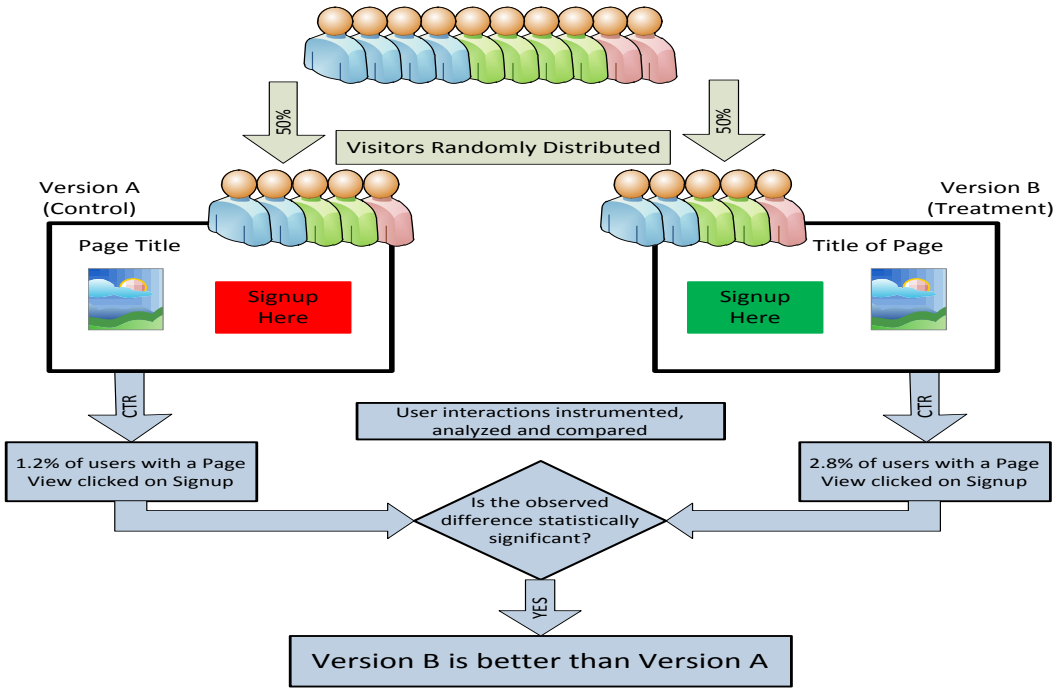


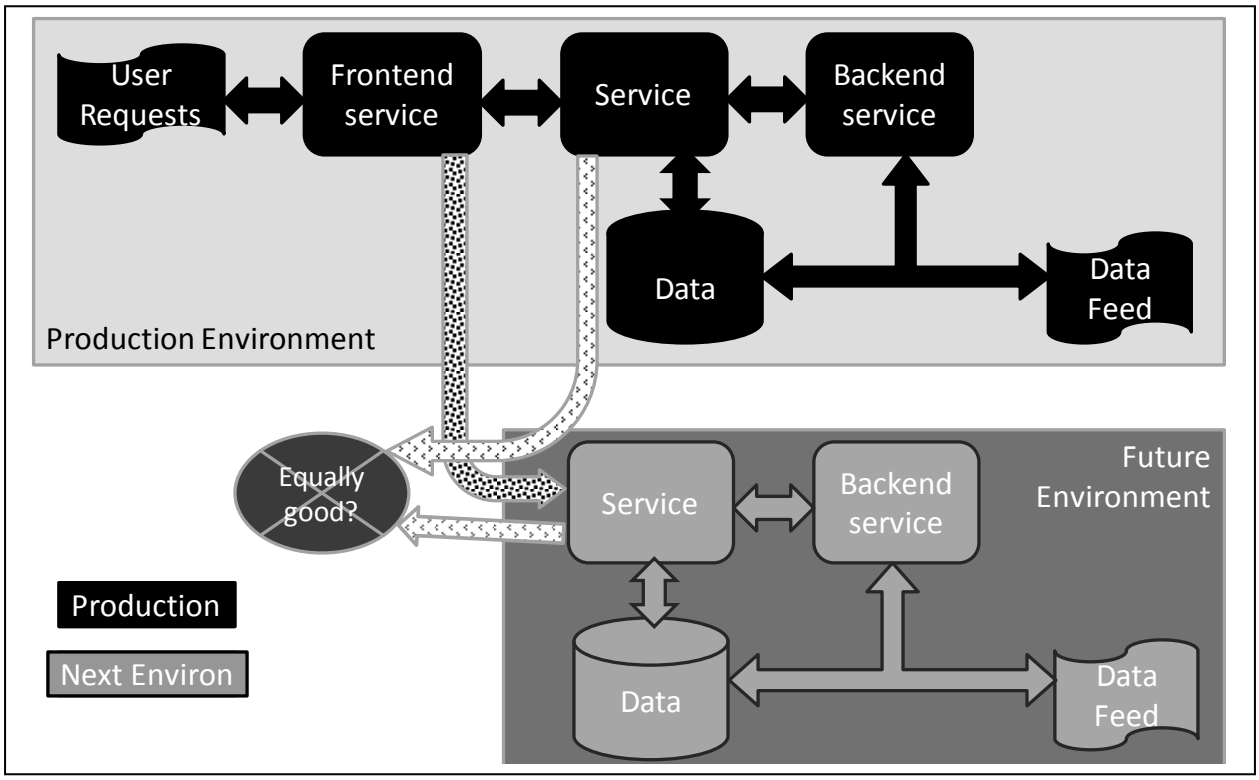
Figure courtesy of Seth Elliot from his SASQAG (Seattle Area Software Quality Assurance Group) April 2011 Talk Testing in Production - Your Key to Engaging Customers

(CTR = Click Thru Rate)

You can also consider mixed-mode testing (have a current version and next version of the same service running) for A/B testing by looking for no statistically relevant differences. Depending on the features or changes in the new version, you may expect to see one or more statistically relevant improvements. You can be looking specifically at CPU usage, Disk usage, I/O, etc. You also may look at the traditional user engagement metrics (Click-through rate, time on page, etc.), as a decrease in these for new version might indicate a possible problem (i.e., if the service is not responding in a timely fashion, it can cause users to click away from the page). You can then decide, based on the data, if the new version is an improvement worthy of complete roll out and replacement of the current version.

A/B testing requires enough instrumentation to tell if you are doing better or worse than before and especially the ability to react quickly when it shows the new treatment is worse than the old control. Typically the treatment is discontinued as soon as it is shown to be statistically significantly worse. To obtain statistical significance there must be sufficient usage of the both the A and B (or control and treatment) versions. When the new treatment is shown to be better for a sufficiently long period of time, it can replace the control treatment as the standard version.

An easier method is to just shadow the production environment with a new environment and fork the requests and compare the replies and other data to see if the new environment performs equally or better.



2.5 Live Capacity Testing (fake traffic)

Live Capacity Testing is done only after your planning indicates you should be able to handle the capacity and smaller scale tests have been done to provide some level of credence to the planning models. However, planning models and the full scale are rarely identical and thus, after small tests and capacity model indicate it should work; the service should verify it actually does work.

The first goal of a live capacity test is to not impact users. This goal must remain paramount by constant monitoring of Service Level Agreements (SLAs) and throttling back or discontinuing the test even before an SLA is broken. Two major safety mechanisms are test auto-throttling and product throttling. In test

auto-throttling the load generation tool consumes real time monitors and will automatically pause or stop the load as the test approaches SLA or Key Performance Indicator (KPI) thresholds. With product throttling, the servers have technology to throttle capacity traffic if latencies increase significantly. Throttling can be by discarding low value traffic (e.g., Bot traffic). Throttling can also occur (but less desirably) by producing lower quality results (e.g., older cached data, instead of the freshest data).

A goal of live capacity testing is to verify business continuity scenarios in production at or beyond existing observed peak load while meeting SLA (uTest 2011). Typical components of an SLA include availability (request completion), request latency (e.g., average and percentiles), as well as requests per second. For an example goal, let N be the recently measured peak of requests per second. Assume you want to allow for projected growth of fifteen percent (15%) and you want to accomplish all of this during a Business Continuity (BC) event when one of your four Data Centers (DC) is down. With four DCs and one DC down, each remaining DC must have 1.33 capacity needed, so capacity test traffic would be 1.15 * 1.33 = 1.53N

Live capacity testing allows physical asset verification (network gear, DC power, etc.), validation of the service in new datacenters, and optimizing utilization of existing equipment via proper balancing. In prior tests, Bing has observed overheating when everything ran full throttle, or exceed maximum power input for the DC from utilities.

A way to do capacity testing is having a controller drive each of the clients across all DCs. The load is generated within each datacenter to avoid cross-DC network utilization (unless that is what needs to be tested). The controller pulls data from monitoring system, auto-throttling based on SLA. Are you providing the traffic using your products' APIs or also via emulated users using browsers? A capacity test passes when traffic goals are reached within SLA. You also need to consider the impact of your capacity test on partners that your service uses. Do you want to fake the partner traffic or really incur potential costs of capacity testing your partners also?

In planning a capacity test, first determine the overall goal. Examples goals include:

- Set targets based on your service history, predicted growth, required headroom buffer, and Business Continuity goals.
- Verify capacity of default experience
- Verify capacity of ancillary and partner services.
- Verify capacity of new features
- Verify a Business Continuity scenario
- Verify for a period of time, e.g., 4 hours or 24 hours

Second, determine the correct mix of traffic. This is non-trivial as there are many dimensions to consider. Examples of traffic types to mix:

- default traffic,
- Experimental traffic, e.g., future traffic for new feature or service
- markets, e.g., North American, European, Asian
- Geographic, e.g., Central US, Southern US
- patterns, (e.g., Distributed Denial of Service , DDOS, attacks, or typical calling patterns within the service)
- distribution,(e.g.,. cheap operations (like query) vs. expensive (like update))
- temporal (traffic is time dependent)
- cache hit ratio (traffic does or does not hit cache)

Final setup step is to distribute necessary data such as previous sanitized or masked logs of users' interactions with the service, or distribute example inputs to capacity clients.

Before running a capacity test, notify Designated Responsible Individuals (DRI) and capacity partners, ensure site is in a good state, and establish communication, for example via instant messaging or conference call. Then start load generation tool and ramp traffic slowly (until shown peak load is handled, so can throttle back with less risk if peak load not currently handled within SLA). Eventually try different load ramping methods such as spikes or steps. As stated previously extensive monitoring is critical. Stop the test when limits are exceeded, goals are met, or time runs out.

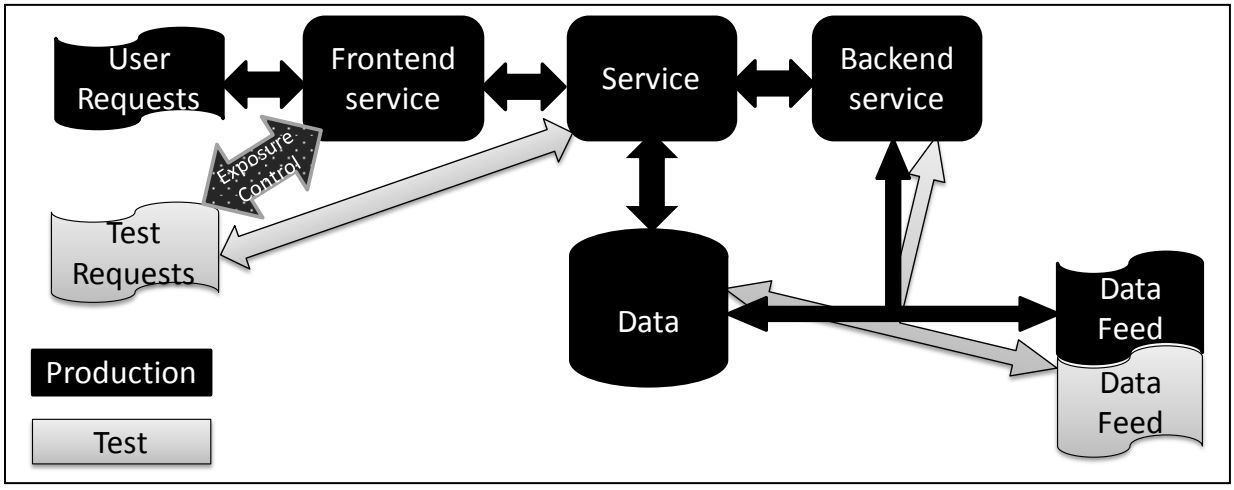
2.6 Tracers

Another method of Testing in Production is verifying correct data flow through the system. While data flow testing of the service in a test environment may show it should work, there is always the possibility of operational and configuration error such that it doesn't work in production. Thus, constant monitoring via injection of test data into back-end data feeds is another critical factor in verifying correct operation of a service. Like synthetic transactions, the data must be marked or handled such that it doesn't impact billing or services metrics related to real traffic.

With observation of the traces at various points throughout the data flow, any data dropped or misrouted instead of correctly transferred can be quickly detected.

For example, in a shopping service you could inject test products, test offers to sell products, and test reviews of products. Monitor injected feeds to see if the product is found in the product authority database. Monitor to see if the offers to sell and review feeds get classified correctly. Monitor if the product matching service matches product offers and reviews to products in the system. Finally, monitor if the user sees product when they search for it; if review search pages display reviews from the feed; and if product details page for a given product contains the matched offer and review information.

In testing tracer data you must consider whether you want or need to control the exposure. Decide if test products can be seen by the general public or only special logins, network addresses, etc.



2.7 Fault Injections

All large systems will have failures. They may be due to hardware failure, software failure, or environmental failure. While failure prevention is encouraged, it is never enough. Thus the system must be resilient enough to recover from failure, and recovery oriented computing (Brown 2001) should be considered.

Fault injection and resiliency testing should be done first in test labs, but that is not sufficient. Again, configuration, operations, specific hardware, etc. may come into play to make live production unique and different from a production-like environment.

For testing in production, we need to verify that the production system is indeed resilient to failure by deliberately inducing failures where the system is relatively reliable. Many services find that some failures, e.g., disk drive failure, are so common (multiple daily occurrences in a large data farm), that no special fault injection is needed. The failure recovery is frequently tested due to the unreliability of certain aspects of the system.

For the relatively reliable system parts, fault injection should occur. It can be planned, but ideally the system can handle random failures as is done by Netflix's Chaos Monkey (Ciancutti n.d.). You may limit the faults to those seen previously and based on previous frequency as a fault operational profile initially. Eventually you should be able to test for all fault types, although frequency of each one remains a critical decision. Commercial devices exist for inducing network data corruption, network latency, and network routing issues. Similarly you can have various types of controlled power failure. Via software you can cause loss of a process, a machine, a service, or potentially the entire DC.

Fault Injection may cause other unexpected failures or potentially a cascade of failures (Amazon 2011). Verifying you can control failure cascades and diagnose unexpected failures are just additional service capabilities you must have that Fault Injection helps verify.

3. Conclusions

Starting from Offline testing in a production environment, most services have added monitoring including asserts, and usually at least limited synthetic transactions. A/B testing experiments have provided proven value that more sites are adopting. But as numerous service outages have shown, more must be done. Architecting the service for Self-Test and making virtually all tests available as Synthetic transactions is required. Part of architecting a service for test in production is dealing with test data in the live environment, whether from synthetic transactions, tracers, or other means. Finally, all type of testing including performance, capacity, resiliency and stress tests must be engineered to work while the live system is running through a combination of service code, monitoring code, and test code.

Examples of companies using Test in Production surviving major failures is growing and conversely there are too many examples of services not handling major failures and not fully testing in production.

Summary Table

Approach	Risk of Causing issues	Exposure Control	Data Control
Asserts	Medium	Optional	N/A
Monitoring	Low	N/A	N/A
Offline	Low	N/A	N/A
Controlled Experiment	Medium	Required	Useful
Live Capacity	High	Useful	Required
Tracers	Medium	Useful	Useful
Fault Injection	High	Optional	Useful

3.1 Acknowledgements

Greg Veith has exemplified how to do Live Capacity Testing. Ken Johnston and Seth Eliot continue to expand my understanding of Test in Production.

References

(MarketingExperiments n.d.)Amazon. *Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region.* 2011. http://aws.amazon.com/message/65648/ (accessed 6 21, 2011).

Brown, A. and Patterson, D. A. "Embracing Failure: A Case for Recovery-Oriented Computing (ROC)." *High Performance Transaction Processing Symposium.* Asilomar, CA, 2001.

Ciancutti, J. *5 Lessons We've Learned Using AWS.* n.d. http://techblog.netflix.com/2010/12/5-lessons-weve-learned-using-aws.html (accessed 6 1, 2011).

Eliot, S. *Feeling TiPsy…Testing in Production Success and Horror Stories.* 12 14, 2009. http://blogs.msdn.com/b/seliot/archive/2009/12/14/feeling-tipsy-testing-in-production-success-and-horror-stories.aspx (accessed 6 1, 2011).

Google. *Advanced A/B Testing.* n.d. http://www.google.com/support/websiteoptimizer/bin/answer.py?hl=en&answer=62999 (accessed 6 21, 2011).

Gray, J. "Why do computers stop and what can be done about it?" *Technical Report 85.7, PN87614, Tandem Computers, Cupertino.* 1985. http://www.hpl.hp.com/techreports/tandem/TR-85.7.pdf (accessed 6 1, 2011).

Hamilton, J. "On Designing and Deploying Internet-Scale Services." *Proceedings of the 21st Large Installation System Administration Conference (LISA '07).* http://www.usenix.org/event/lisa07/tech/full_papers/hamilton/hamilton_html/ (accessed 2011/06/01), 2007. 231-242.

Infosec. *Amazon AWS goes down, takes several others along with it.* 04 21, 2011. http://infosecindia.com/2011/04/21/amazon-aws-goes-down-takes-several-others-along-with-it/ (accessed 06 21, 2011).

Johnson, K.N. *Performance testing in the production environment.* n.d. http://searchsoftwarequality.techtarget.com/answer/Performance-testing-in-the-production-environment (accessed 6 1, 2011).

MarketingExperiments . *A/B Split Testing.* n.d. http://www.marketingexperiments.com/improving-website-conversion/ab-split-testing.html (accessed 06 21, 2011).

Meszaros, G. "Delta Assertion." In *xUnit Test Patterns*, by G. Meszaros, http://xunitpatterns.com/Delta%20Assertion.html (accessed 2011/06/21). Meszaros, G.: Addison-Wesley, 2007.

Michelsen, J. *SOA Testing Best Practices: Complete, Collaborative, and Continuous.* n.d. http://advice.cio.com/john_michelsen/soa_testing_best_practices_complete_collaborative_and_continuous (accessed 05 1, 2011).

Nguyen, C. *Facebook Website Down?* 12 2010. http://www.ubergizmo.com/2010/12/facebook-website-down/ (accessed 6 21, 2011).

Rathi, M. *The Support Authority: Why testing in production is a common and costly technical malpractice.* 02 02, 2011. http://www.ibm.com/developerworks/websphere/techjournal/1102_supauth/1102_supauth.html?ca=drs- (accessed 06 01, 2011).

Rigor. *Continuous Deployment - Functional Testing in Production.* 1 7, 2011. http://rigor.com/blog/continuous-deployment-functional-testing-in-production.html (accessed 6 1, 2011).

Soasta, Inc. "CloudTest® Strategy and Approach." 9 27, 2010. http://www.cloudconnectevent.com/downloads/SOASTA_TestingInProduction_WhitePaper__v1.0.pdf (accessed 06 01, 2011).

uTest. *Testing the Limits With SOASTA's Dan Bartow – Part I.* 01 2011. http://blog.utest.com/testing-the-limits-with-soastas-dan-bartow-part-i/2011/01/ (accessed 6 1, 2011).

Exploring Cross-Platform Testing Strategies at Microsoft

Jean Hartmann
Principal Test Architect
Office Engineering - Test

Microsoft Corp.
Redmond, WA 98052

Tel: 425 705 9062
Email: jeanhar@microsoft.com

Abstract

The Office Productivity Suite including applications, such as Word, Excel, PowerPoint and Outlook, has now been available for Windows PCs and Apple Macintoshes for many years. During these years, the respective product and test code bases have grown significantly, with increasing numbers of features being added and requiring validation. When validating PC-based products, testers leveraged some of the benefits of the Windows platform including the availability of the .NET framework to implement their tests using managed programming languages. For test execution, they used Windows-supported mechanisms, such as COM/RPC, to communicate in- and out-of-process with the application under test. Thus, when teams needed to deliver related Office products on a different platform, such as the Apple Macintosh, test teams were faced with a dilemma - either attempt to port test cases, together with the required test infrastructure, or create new tests using the platform-preferred development/test environment. Both were time-consuming.

With the advent and rapid evolution of mobile platforms, such strategies are becoming more difficult to justify - implementing test suites from scratch is just too costly and slow. New test strategies, tools and processes are needed that promote the construction of portable tests and test libraries and enable testers to quickly retarget a given test case for different platforms and devices. This approach is particularly valuable when validating the common or 'core' application logic of each Office product for different platforms and devices, resulting in a more consistent level of quality for core functionality. It also gives test teams more time to focus on validating those product features that are unique to a specific platform or device.

This paper chronicles our ongoing exploration of platform-agnostic testing strategies during the current shipping cycle. It highlights the challenges that we have faced so far and attempts to illustrate and emphasize key concepts of our work using examples.

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

Companies, such as Microsoft, are embracing mobile technologies in the form of tablets and smartphones at a rapid pace. For the Microsoft Office Division and its suite of productivity tools and cloud services, this means complementing traditional desktop client applications for the PC and Apple Macintosh as well as cloud offerings, such as Office365, with native and browser-based mobile applications for the most popular platforms and devices including Apple's iPhone and iPad, Google's Android, Nokia's Symbian and Microsoft's Windows Phone 7.

To address the need for cross-platform testing, the division could either develop required expertise in-house or outsource the effort to established mobile application testing vendors. For manual testing, groups of testing professionals can be hired - they use a range of mobile devices in real-world conditions and attempt to find bugs. Vendors, such as Mob4Hire [1] and uTest [2], offer such services. Alternatively, third-party vendors including DeviceAnywhere [3] and Perfecto Mobile [4], provide automated testing solutions, which have the benefits of speed and control as well as scalability.

The decision taken by the Office organization was to build up the required in-house expertise based on an automated testing solution. As we embarked on developing such a solution, there were a number of aspects that we needed to take into consideration:

a) **Mobile platforms and devices have significant limitations, when compared to our desktop environments.** When testing in our desktop environments, we take certain things for granted. For example, we assume access to fully-fledged runtime environments, such as the .NET framework, to execute our managed test code and products, operating systems that are capable of supporting multiple processes to execute our regression tests out-of-process and sophisticated communications or marshaling mechanisms, such as COM/RPC, to allow us to interact our target applications' interfaces. These issues all need to be re-examined in the context of cross-platform test execution.

b) **Leveraging existing test automation environment is desirable**. We did not want to re-invent the wheel, when it came to defining a suitable, automated test infrastructure. Instead, we wanted to explore ways of seamlessly extending the power of our *existing* PC-centric test environment to encompass *actual* mobile devices rather than interacting with emulators[1]. We investigated mechanisms that would enable PC-based test clients to efficiently and effectively interact with devices applications. As a result, existing test cases could continue to be executed on the PCs with interaction being redirected at the mobile applications.

c) **Leveraging existing, automated test suites is desirable.** Given that the Office organization has built up extensive test suites for its existing desktop applications, the goal is to determine whether it is more cost-effective to refactor those existing tests for portability and share them across new devices and platforms versus implementing and then maintaining new, platform- or device-specific test suites. The decision hinges upon the amount of *product* code and functionality that can be shared across platforms and devices – if a large percentage of the core application logic is common across platforms and devices, then refactoring for portability is justified. In contrast, if products share little or no functionality across platforms and devices, then it would not be worth the refactoring effort; custom test suites would be more appropriate.

The remaining sections of this paper describe our ongoing investigation into cross-platform testing strategies. In Section 2, we explain how we design for test portability. Section 3 outlines our enhanced test infrastructure built upon our existing test tools. For Section 4, we highlight

[1] Based on the fact that the current emulators provided by platform and device manufacturers are not of the required quality and sophistication for large-scale, automated, regression testing of enterprise applications.

some process- and product-related issues related to this test initiative. In Section 5, we summarize our ongoing investigation by outlining an ongoing feasibility study that brings together our findings and concepts as a set of sample test cases and prototyped test tools. Finally, Section 6 presents conclusions and next steps in our effort to deploy this approach throughout the Office division.

2. Designing for Test Portability

Many thousands of automated, PC-based test cases have been written by Office teams over the years. The code manipulates a product's UI (User Interface) or OM (Object Model), making each test useful only in its original PC-based application context or *application-context-specific*.

Portable tests, on the other hand, should **not** be application-context-specific! The test code should not exhibit platform- or device-specific details and instead emphasize 'test intent rather than implementation'. The associated test libraries or *task libraries* as they are referred to within the Office division, need to support portability by exposing an abstract set of interfaces to the test code and encapsulating the nuances of the different application contexts (application/platform combinations). The following section describes the basics of designing such test cases and libraries.

2.1 Task Libraries

A key challenge for test portability lies in the design of well-abstracted task libraries for each product. Whether these task libraries are being implemented for new products or refactored based on existing libraries, appropriate abstract interfaces need to be defined. These interfaces need to reflect the application's common and unique properties for each context, that is, platform and device. Consider the Word word-processing product and some of its properties; we can define three new abstract interface classes:

```
interface IWordApplication
interface IWordDesktopApplication : IWordApplication
interface IWordBrowserApplication : IWordApplication
```

The first interface encapsulates the *common* properties of the application across all target platforms and devices. For example, in the Desktop and Browser versions of the Word application, testers can manipulate the canvas programmatically. So, the `WordCanvas` property should be declared in the `IWordApplication` interface to indicate that all implementers expose a document canvas to the tester. If necessary, a new `IWordCanvas` interface can be defined to abstract the canvas implementations and define public canvas tasks for all application contexts.

The other two interfaces reflect the *unique* properties of the application on specific platforms or devices. For example, only the Desktop version of Word supports COM Add-ins, so a `ComAddIns` property might be exposed in the `IWordDesktopApplication` interface, but not the `IWordApplication` or `IWordBrowserApplication` interfaces. Any changes to the base class `IWordApplication` with its common Word properties are then automatically inherited by the application-context-specific interface classes.

In the case of existing task libraries, once these abstract interfaces have been defined, the existing task library classes need to be modified to implement the new interfaces and the test cases are updated without the test code requiring modification. For example, for an existing Word task library class `DesktopWordApplication`:

```
/// <summary>
/// Class DesktopWordApplication is the base class that defines
/// tasks available for the Desktop Word application.
/// </summary>
public class DesktopWordApplication
{
    /// <summary>
    /// Gets the Word canvas.
    /// </summary>

    public WordCanvas Canvas
    {
        get
        {
            // return canvas...
        }
    }
  // Other Desktop Word Application members.
}
```

We now define a new interface `IWordApplication` containing only or all the public signatures from the existing task library class:

```
/// <summary>
/// Interface IWordApplication is the base interface that declares all
/// tasks available for the Word application.
/// </summary>
public interface IWordApplication
{
    /// <summary>
    /// Gets the Word canvas.
    /// </summary>
    WordCanvas Canvas { get; }
    // Other common Word Application members.
}
```

The existing task library class now implements the new interface:

```
public class DesktopWordApplication : IWordApplication
{
    /// <summary>
    /// Gets the Word canvas.
    /// </summary>
    public WordCanvas Canvas
    {
        get
        {
            // return canvas...
        }
    }
    // Other Desktop Word Application members.
}
```

The refactoring of the existing task libraries also presents opportunities for cleaning up any test code that explicitly references context-specific logic. This logic needs to be moved out of the test case and into the new, abstracted task libraries as an appropriate property, method or class.

2.2 Test Cases

With the task libraries structured and exposing a set of suitably abstracted interfaces, testers are ready to code against those interfaces. In the case of refactored task libraries, an additional step is needed to ensure that the same test cases can be executed against an application running on multiple platforms and devices. The test code needs to be updated by replacing each class Type with the Type of the highest, relevant interface in the inheritance hierarchy.

For example, consider an existing test case *Word Canvas Bold Test*. The test case logic declares a variable, `WordCanvas canvas`, and accesses `WordCanvas` members to automate the appropriate user tasks. Assuming that the `WordCanvas` class is now extending an abstract interface `IWordCanvas` with the same public member signatures, the test case code needs to be modified, so that the `canvas` variable is declared as Type `IWordCanvas`. Then, successive invocations of members on the canvas object will execute as they did before. Also, if a `NewContextWordCanvas` class is added to implement the IWordCanvas interface, the modified test case can be executed in the new context without further modification.

Existing Test Case	Updated Test Case
`WordCanvas canvas = …;` `Log.VerifyTrue(canvas.Selection.Bold, "Comment");`	`IWordCanvas canvas = …;` `Log.VerifyTrue(canvas.Selection.Bold, "Comment");`
`NewContextWordCanvas canvas = …;` `Log.VerifyTrue(canvas.Selection.Bold, "Comment");`	`<Same as above – no modification required.>`

Test cases and their task libraries are now ready to be used. The former will only contain code that references a set of abstract interfaces; they do not contain any platform-specific code. For that, an additional step is needed during test execution in which the test code is compiled and linked against the task libraries and in particular, *task library implementations* – these represent code snippets for exercising product interfaces on specific platforms and devices.

2.3 Task Library Implementations

These code snippets, together with the automation framework and communications mechanism described in Section 3, provide the third piece of the test portability puzzle. The snippets represent sequences of C# calls, which the test infrastructure maps to calls against the native application interfaces on a specific device/platform. The infrastructure then marshals those calls between the PC executing the test case and the target device. Successfully mapping and marshaling these calls is a key aspect as well as risk to the test initiative and will be discussed in more detail in Section 5. At this time, we are prototyping simple test cases, task libraries and task implementations for select Office applications on Android and iOS devices in order to explore the pros and cons of different approaches to mapping/marshaling problem.

2.4 Application Context

Devoid of any application context and without any direct class references, the portable test cases containing the abstracted interfaces now need to be bound to specific task library implementations at runtime. To achieve this, testers need to define a *test profile*, that is, specify additional test case metadata representing the platforms or devices on which they need to run. In Office, such data is captured in a *scenario* file. The test execution harness (discussed in detail in

Section 3) then provides a mechanism to load a value indicating the application context in which the test case should be executed. The mechanism then interprets the application context value and creates a factory object to a global data store. This factory object is retrieved at runtime to instantiate the appropriate task library implementation for the specific platform or device.

3. Supporting Test Infrastructure

3.1 Existing Test Infrastructure

Traditionally, Office test teams have relied upon a sophisticated, in-house set of test tools. A PC-based automation framework known as *Motif* and a distributed test management system called *Oasys/Big Button* are the two key tools for executing tests in batch. *Oasys,* or *Office Automation System*, a distributed test management system, comprises of a 'master' controller that interacts with a number of 'slave' PCs on which tests are executed one at a time. The interaction between master and slaves is maintained via agent software known as *OAClient*, resident on each of the slave machines. Also resident on each slave machine is the Motif automation framework and the Office application under test. The C# test code and associated task libraries are executed with the help of the Motif framework. Figure 1 shows the existing infrastructure with the OASYS server acting as master controller and two slave or OAClient machines.

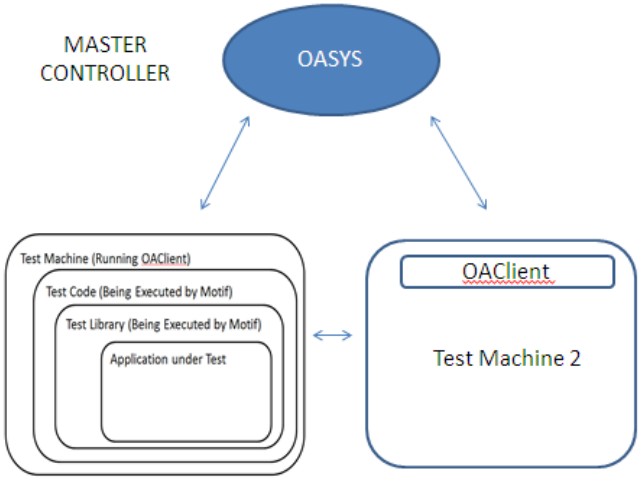


Figure 1: Existing Test Infrastructure

The system makes use of large server farms containing thousands of machines to maximize test throughput – thus the name *Big Button*. Given the scale of past investments in this testing infrastructure and the maturity of these tools, it was desirable to leverage them and not build another dedicated environment from scratch. The challenge, therefore, was how to extend the existing infrastructure to these new platforms and devices and at the same time, avoid porting of infrastructure, such as the OAClients, to the new devices and platforms. The latter would have been difficult due to the restrictive nature of the mobile devices and platforms mentioned in Section 1.

3.2 Extensions

Figure 2 illustrates how we are extending the existing test infrastructure. The extensions comprise of additional hardware and software attached to the slave or test machines. On the hardware

side, we envisage that for each PC-based slave or test machine, there is an attached Windows PC or Apple Macintosh acting as a *device host machine.* Each device host machine is, in turn, attached to one or more devices, for example, smartphones or tablets via USB cable. Device hosts are *required* for flashing and deploying applications onto the various devices. During test execution, whenever test or product failures occur, the machines are needed for collecting and passing back logging data to the test machine to aid debugging.

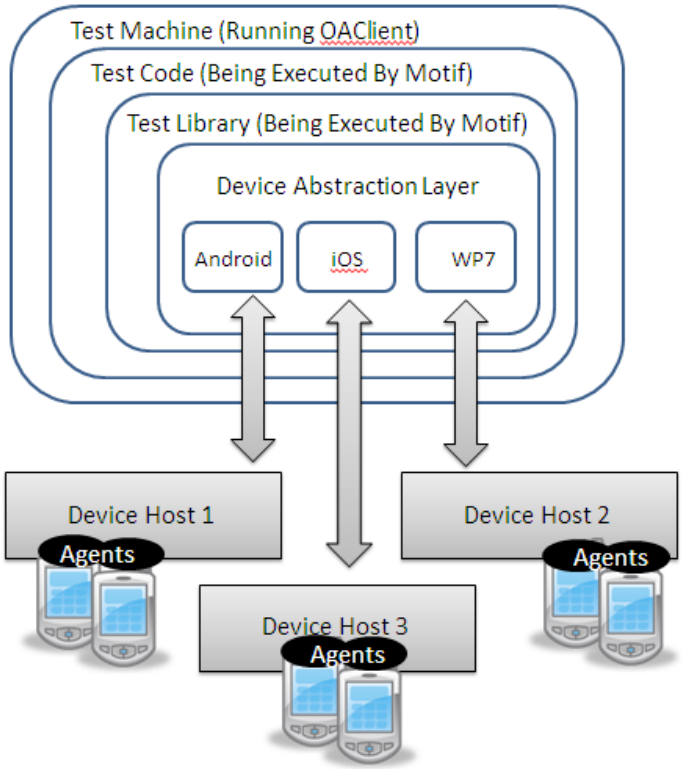


Figure 2: Extensions to Test Infrastructure

As each platform/device has specific installation and deployment requirements, a set of device management APIs, depicted in Figure 2 as the *device abstraction layer,* is required. Teams call into these service interfaces, when defining configuration scenarios prior to their test automation runs. The goal of this layer is to define a common mechanism for tasks, such as setting up and tearing down target applications prior to executing the body of the test case. During the test set-up and teardown phases, the test machine will send instructions to the target device via the attached device host and USB connection to deploy and remove applications.

Another key piece of software relates to the communications between the device and the OAClient machine. During test execution, this communications channel is responsible for 'remotely' stimulating the application under test on the device and receiving responses as well as log data. In essence, it forms a 'channel' between the PC executing the test case and the target device under test. We envisage communicating directly with the target device via wireless (WiFi) rather than via USB and attached device host. Although USB-based communications is faster and more reliable, it would require some brands of mobile devices to be 'jailbroken', which is illegal.

Building such a channel requires supporting infrastructure on both the PC test machine and each target device. For that, we decided to implement '*agents'* on each type of device, shown in Figure 2 as black bars. Corresponding software is available on each test machine to interact with the different agents. Each agent, which represents a minimal amount of code, helps to establish a bi-directional TCP/IP-based, that is, socket-based communications between the target devices and

test machine[2]. This channel enables the native application (APIs) running on the devices to be stimulated from the PC side by passing XML messages containing API and related type information to them via the agent.

No additional test infrastructure resides on the device. It is paramount that these agents are kept as lightweight as possible, because using any significant device memory and processor resources means that the application would not be tested under the same conditions as when it is actually being deployed.

3.3 Supported Test Scenarios

The above approach was developed to support the following usage scenarios:

a) **Single-client scenarios** – tests need to validate features of an individual Office application on a specific mobile platform or device. Scenarios, for example, include the application opening a file on SharePoint, then editing, saving and closing the file, then repeating that operation sequentially across different platforms/devices.

b) **Multi-client scenarios** – these are tests that validate the features of an individual Office application, which supports concurrent interaction between users on multiple platforms/devices. For example, two users of Word's co-authoring feature may be editing and reviewing a common document residing on SharePoint. Another example is the Lync communications application with multiple users around the world, each with their Lync client, interacting via voice/video.

4. Related Issues

Whenever new testing strategies and tools are introduced into an existing software development process, there are process- and product-related issues that are impacted. Once portable test collateral has been created, a process-related issue is how to maintain these portable tests over time. Rather than each platform team maintaining its own independent test suite, the responsibility for test maintenance is now shared across platform teams. Teams need to have an agreement or understanding in place, so that whenever product or test failures occur on a specific platform, the team owning that product/platform combination takes action. This action could result in updates to product, test cases, task libraries or task implementation.

Creating a common test suite across platforms may also lead to a redistribution of available testing resources with one part of the test team maintaining this shared test suite to validate the core functionality of the application against all platforms with the rest of the team conducting user interface testing of the product, specific to each platform. This assumes that common or 'core' product functionality has been identified and can be tested separately[3].

Product-related risks include the correct mapping of application interfaces to their corresponding task implementations and the marshaling mechanisms between the test PC and the 'remote' devices. Given that task implementations must be available in C# and Office mobile applications, are currently implemented in a variety of programming languages including Java and C/C++, 'stub generation' strategies and tools are needed that keep the two worlds in sync. Moreover, some applications expose their test interfaces using Microsoft's COM/RPC. If tests are to be maintained in C# on the test machine, how do we communicate with these COM-based interfaces

[2] At the time of writing, we are investigating different IP-based communications protocols including HTTP.
[3] For example, it may have been identified as part of a product refactoring effort that has led to use of the model-view-controller (MVC) design concept.

on the remote devices? Do we port the COM/RPC stack or redefine the test interfaces, keeping them simple and enabling a simpler marshaling mechanism or protocol?

5. Feasibility Study

In order to prototype and validate our test code and test infrastructure, we are currently conducting a small feasibility study. For validating our test infrastructure improvements, we established a private or miniature test lab containing a scaled-down version of our typical Oasys test environment. This 'minilab' included two Windows-based OAClient machines and a representative set of target platforms and devices including Apple equipment (Macbook with tethered IPod and IPad) as well as Android- and WP7-based mobile smartphones and tablets. At the time of writing, we are investigating the different deployment schemes for installing the operating systems and native applications as well as the custom, device-side agents that we need for device communication. This will enable us to reach a point where sample tests can be executed on the target devices. For validating our test code and task library improvements in the context of the supported scenarios, we are implementing a set of test cases representing: a) single-client, interoperability scenarios focused on file I/O where a set of common Office document types being loaded, saved and compared on each platform and device and b) multi-client scenarios that include the OneNote co-authoring scenario.

6. Conclusions and Future Work

In this paper, we described the latest findings from our ongoing investigation concerning the topic of cross-platform testing. We highlighted our motivation for this effort and outlined the key areas in which we made improvements, namely test code and associated task libraries, test infrastructure and test processes. Our exploration is far from over and we still face a number of technical hurdles including the implementation of logging and debugging support, etc. Future work will also entail pilot studies with select teams that will result in best practices and guidance concerning the design of portable tests and task libraries.

7. Acknowledgements

I would like to thank Curtis Anderson, Principal Test Manager of Office Engineering Test, Tara Roth, Corporate VP of Office Shared Services, Mary Smith and Rob Daly, Principal Test Managers for their ongoing support. I also want to express my gratitude to all cross-platform testing initiative team members for their contributions, support and discussions concerning this work. In particular, I would like to thank Forrest Dillaway, Julio Lins, Richard Wright, Jay Daniels, Ashley Tran and Jeffrey Weston. Thanks also go to my reviewers whose invaluable feedback and comments are greatly appreciated.

8. References

1. http://www.mob4hire.com
2. http://www.utest.com/what-we-test/mobile-application-testing
3. http://www.deviceanywhere.com/
4. http://www.perfectomobile.com/

A Distributed Randomization Approach to Functional Testing

Author(s)
Ilgin Akin & Sam Bedekar

Abstract

Traditionally, much of software testing focuses on controlled environments, predefined set of steps, and expected results. This approach works very well for tests that are designed to test targeted functionality of a product. However, even when test automation results are all green, have you wondered what other bugs might be hidden in the product? Have you hit a point in your test automation where you felt like you are not finding new bugs anymore but only regressions?

These are some of the questions we had been thinking about at Microsoft that led to the AutoBugbash project. A standard bugbash is a focused amount of time where the whole team gets together in a room and pounds on the product. Bugbashes are a highly effective way to find a lot of bugs. There are more eyes on the products and people tend to do a lot of random actions that may not be part of their day-to-day structured testing. However, bug bashes are expensive in terms of manual labor.

AutoBugbash is a form of Bugbash automation that incorporates the elements of decentralization and randomization. There are two main components to this approach. The first component is a set of standalone test clients that autonomously take actions on their own and verify expected behavior locally and log their observations with minimal state checking and no predefined script. The second component is a post-run component called the reconciler which reconstructs the sequence of events by parsing logs and matching events to actions. With this paper, we will describe how the AutoBugbash project helped uncover crashes and other hard to find bugs within the Microsoft Lync product.

Biography

Ilgin Akin has been working in the Unified Communications group at Microsoft as a Software Development Engineer in Test since 2006. Prior to Microsoft, she has worked at Pitney Bowes Corporation as a Software Developer.

Sam Bedekar is a Test Manager at Microsoft. He has worked in the Unified Communications space for nine years. Sam is very passionate about test and the impact it can have on product quality and the industry. Sam believes that Test Engineering has vast potential to grow and is excited about leading efforts to bring the state of the art forward.

Acknowledgements

Bo Yang, David Moy, Jack Banh, Matt Mikul

1. Background of this Case Study

Microsoft Lync 2010 is an Instant Messaging, Presence, VoIP, and Collaboration client that enables remote meetings & communication scenarios. In the context of testing, Lync provides its own set of unique challenges and environmental factors, such as:

1. Most scenarios are distributed across multiple client endpoints/machines.
2. Scenarios can be run on a desktop client, IP phone client, mobile client, or web client.
3. Environmental factors such as operating system and Microsoft Office version can affect the test results.
4. A test case can be written at the user interface (UI) layer, or directly on top of the object model layer.
5. The implementation of the business logic is asynchronous by nature – multiple operations/actions can be sent to the server with responses arriving later. A transaction (or scenario) may involve multiple clients to complete.

1.1 Sample Test Case

One typical scenario in Lync is establishing a call between two endpoints. The basic steps for this test case are:

1. Endpoint-A makes a call to Endpoint-B.
2. Endpoint-B accepts the call.
3. Verify the call is connected on Endpoint-A.
4. Verify the call is connected on Endpoint-B.

In this setting, three entities come from those steps: the controller (or driver), the Endpoint-A, and the Endpoint-B. In a typical implementation of this test case, each endpoint is running on its own machine, while the controller may be running on a separate machine, or on either endpoint machine. In both cases, the three entities communicate remotely. These entities are a common theme throughout our test portfolio.

1.2 Testing Meetings

To give additional context on meetings, a meeting is a virtual conference or meeting room.

1. The Lync meeting room will have several Modalities – Instant Messaging, Audio Conferencing, Video Conferencing, Whiteboard, Application Sharing, PowerPoint sharing, and Handouts.
2. The meeting will have several participants – each joined user is a participant
3. A user may join from multiple endpoints – for example, a user may join audio conferencing from their phone, and application sharing from their desktop PC.
4. Users fall into various permission categories – there is one organizer, multiple presenters, and multiple attendees. Each permission category determines which actions a user is allowed to perform.
5. Each participant, based on role, can interact with various Modalities.

1.3 Examining a Test Portfolio

To test Lync, we needed a diverse set of test cases comprising an overall test strategy. We call our set of test cases a *test portfolio*. A traditional test portfolio consists of test cases of various *test targets* and *test*

techniques. A test target is a strategy to test a particular aspect of a product. For example, functionality, performance, and stress are all test targets. A test technique is a method for testing a particular target. For example, functional tests and beta tests are both techniques that can be used to test functionality. A technique may be classified as automated or manual. An automated case has several advantages – low cost of execution, consistency in methodology. A manual test case has a different set of advantages – appropriate randomization, real-time verification, as well as advanced heuristics for verification, among other advantages.

1.4 A Sample Test Portfolio:

Test Target	Test Technique	Type
Functionality	1. Functional Automation	Automated
	1. Internal Usage of the Product	Manual
	2. Customer Beta Testing	Manual
Performance	1. Time Measurements in a Lab Controlled Environment 2. Time Measurements from user usage data	Automated & Manual
Stress/Scalability	1. Lab Environment stress 2. User Model based scalability	Automated

1.5 The "Bug Bash"

Over time, a technique of manual testing evolved that we called the *bug bash*. The idea was to have a set of folks come together to explore and find bugs in the product in a fixed set of time. The technique was manual in nature and focused on volume of bugs in a short burst. The directive is to "test at will." There are a few aspects that make a bug bash successful:

1. Real Time Decisions – As bug bashes are either unscripted or have broad generalized scripting, testers will often choose their actions. This adds an element of randomization when bringing several testers together.
2. Avoid real time verification that may slow down or change the scenario – Folks would tabulate issues and record bugs later.

This technique was very effective as it found bugs that spanned many test targets. Functionality bugs, performance issues, and stress related bugs are often exposed through bug bashes.
Over time, the bug bash became a staple of our test portfolio. It was a great supplement to many of our other test techniques. As it became a first-class citizen, we started discussing the possibilities of automating the bug bash. We realized the irony of attempting to automate a technique which brought benefits by being manual & randomized. We broke down the components of a bug bash and examined each one to see how we could automate it. Along with that, we had a few clear goals:

1. Repurpose existing automation – this was a clear goal to avoid duplication of effort.
2. Allow cross-platform test cases
3. Simulate a real "meetings" bug bash – allow each simulated user to perform their own actions
4. Test without modifying the meeting cadence – though this is test automation, it should simulate the speed and validation model of a manual bug bash

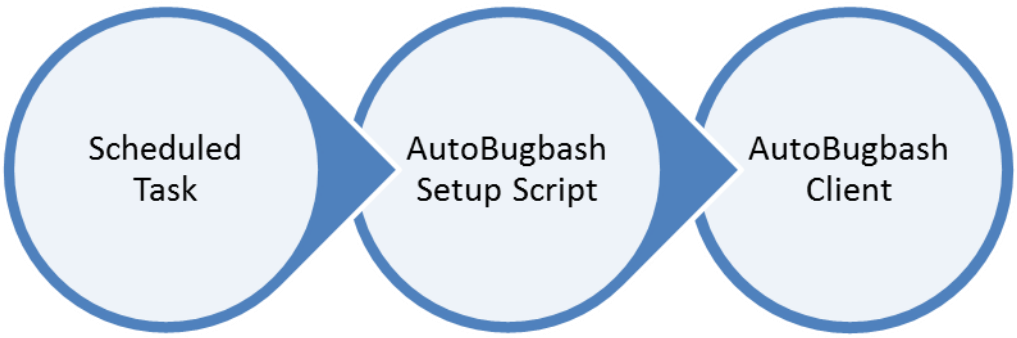
All of these became factors in our design for the meeting auto-bug bash. The meetings operating model was as follows:

1. Users can leave/join as they please – a full-featured meeting is full at 250 users.
2. Users can add a Modality which introduces the Modality into a meeting. Each participant can then choose whether they want to participate in the Modality or not.
3. Typically, when a user performs an action, such as sending an instant message or flipping a PowerPoint slide, other participants see the result of that action (receiving a message or viewing a slide flip respectively).
4. Several users can perform actions simultaneously

2. Architecture of AutoBugbash Framework

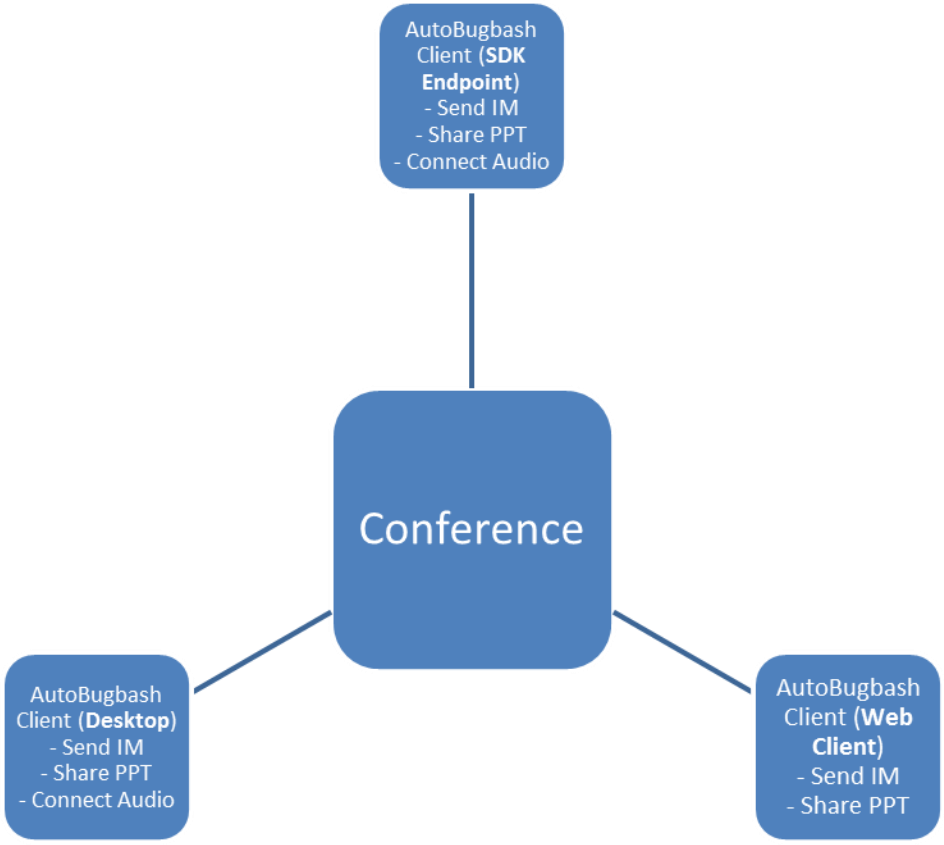
AutoBugbash framework consists of distributed AutoBugbash clients and a Lync meeting that these clients join and perform random actions within a specified amount of time. Examples of actions are sending instant messages, connecting and disconnecting audio, and real-time desktop sharing. During the run, AutoBugbash clients act autonomously and there is no centralized component to control and/or synchronize the clients. After the run is completed, each client uploads its logs and other files, such as dumps, to a shared log repository. Following this, a component we call the reconciler parses the logs, matches each client's actions to events and creates a timeline of the meeting from start to end. The output of the reconciler is a report file that details the run. There are three stages to the AutoBugbash framework.

2.1.1 Pre-Run



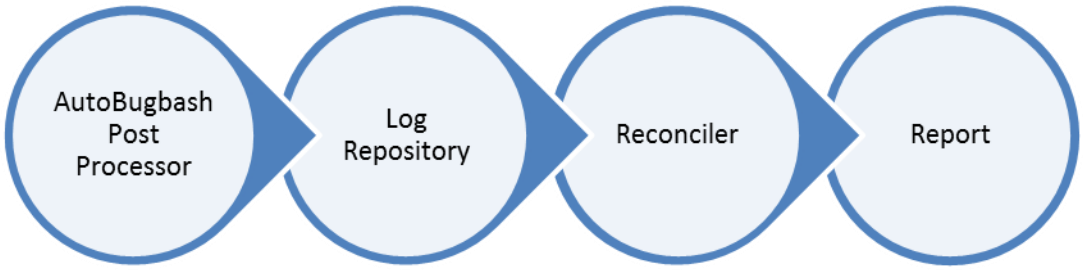
The AutoBugbash pre-run starts by a scheduled task that kicks off the AutoBugbash setup script. The scheduled task is setup on each machine that is going to run AutoBugbash clients. The setup script is responsible for determining the build to use, copying the necessary test and product binaries to the machine and starting up the AutoBugbash client.

2.1.2 Run



During the run, each AutoBugbash client joins a pre-assigned online meeting, and executes random actions until the end of the run. The clients do not wait for each other and do not communicate with each other through. One client might be sending an instant message, while another might be connecting to audio Modality, and another client sharing a presentation. This approach allows the run to be completely non-deterministic. The clients finish the run at a pre-determined time and exit.

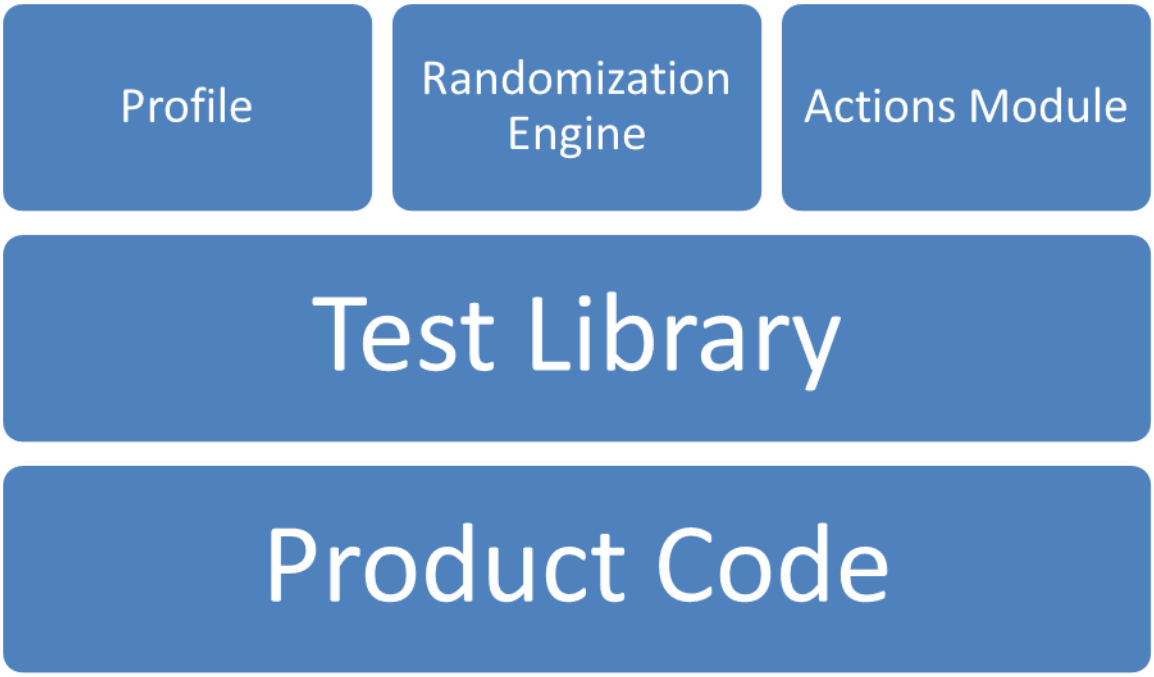
2.1.3 Post-Run



After an AutoBugbash client exits, a post processor script runs and uploads logs and dump files for the client to a central log repository. The log repository is a shared directory, and each client uploads to a separate subdirectory. Following this, the reconciler tool is executed and produces a report file of the run.

2.2 Architecture of AutoBugbash Client

2.2.1 AutoBugbash Client



An AutoBugbash client is a standalone automated test client with AutoBugbash specific components on top of the test library. The main components that sit on top of the test library are the *profile builder*, the *randomizer*, and the *actions* component. The added components are designed to support extensibility and reuse of existing automated test clients for the AutoBugbash framework. Extensibility of the AutoBugbash framework and of the client is designed at three levels. At the reconciler level, any client that adheres to the formatting and rules of the reconciler can plug into the framework.

At the Endpoint Factory level, any endpoint type can be plugged into the framework, as well as at the Action Factory level where multiple sets of actions are supported.
In the following sections, the components will be described in detail.

2.2.2 AutoBugbash Client Call Flow

AutoBugbash client's execution starts with building a profile which the client can follow. The profile uses the Endpoint Factory to create a specific endpoint type, following that a scenario for the AutoBugbash client is created. A scenario is mainly a list of actions that the client can execute based on the type of endpoint. Then the endpoint is initialized, and ready to execute. Throughout the run, the AutoBugbash client runs in a loop, picking a random action from the scenario, executes it and moves to the next action.

2.2.2.1 AutoBugbash Profile

Microsoft Lync supports multiple endpoint types such as UI, OCOM, OCOM.NET, mobile and web endpoints. A Profile describes a endpoint type and a scenario. A Scenario describes selected actions for the client to execute and how the actions are distributed. Each endpoint is capable of executing a subset of all actions. For example, while the Lync desktop client is capable of making calls, the web client is not capable of performing that action.
The Profile component reads an XML configuration file to understand which endpoints are capable of which actions. The XML configuration file also assigns weights to both endpoint types and actions associated with the endpoints. The profile component then selects a endpoint type using the weighted randomization algorithm, and prepares a scenario for the AutoBugbash client to run. There are actions that are not part of the profile, such as creating an online meeting, signing in, and out. Leaving these actions out allows us to have some control over the run, and allow the clients to do more actions. The way the profile picks an endpoint and prepares the scenario is through a weighted randomization algorithm. Each endpoint type and each action has a weight associated with it.
Being able to configure the weights through a centralized configuration file, it is possible to specify the focus of the run for that day. For example, to focus on testing endpoint type A, set the weight of endpoint type A to 80, and endpoint type B to 20, in the configuration file. This way, more clients will pick endpoint A.

```
<GlobalProfile>
 <Endpoints>
  <Endpoint>
   <Type>OCOM</Type>
   <Weight>80</Weight>
  </Endpoint>
  <Endpoint>
   <Type>OCOM.NET</Type>
   <Weight>20</Weight>
  </Endpoint>
  <Endpoints>
 </Endpoints>
</GlobalProfile>

<IndividualProfile>
  <ActionMix>
   <SupportedEndpoints>
    <Type>OCOM</Type>
   </SupportedEndpoints>
   <Actions>
        <Action>
 <Name>AudioConnectAction</Name>
    <Weight>200</Weight>
   </Action>
   <Action>
<Name>AudioDisconnectAction</Name>
      <Weight>200</Weight>
    </Action>
   </Actions>
  </ActionMix>
</IndividualProfile>
```

2.2.2.2 Randomizer

The Randomizer component is a simple component that makes use of the Action class. Any class that extends the Action can be passed to the Randomizer and can be randomized. The Randomizer component provides both standard and weighted randomization. It can both generate a scenario which is a list of randomized actions, and it can also return a random action from a given scenario.

2.2.2.3 Actions Module

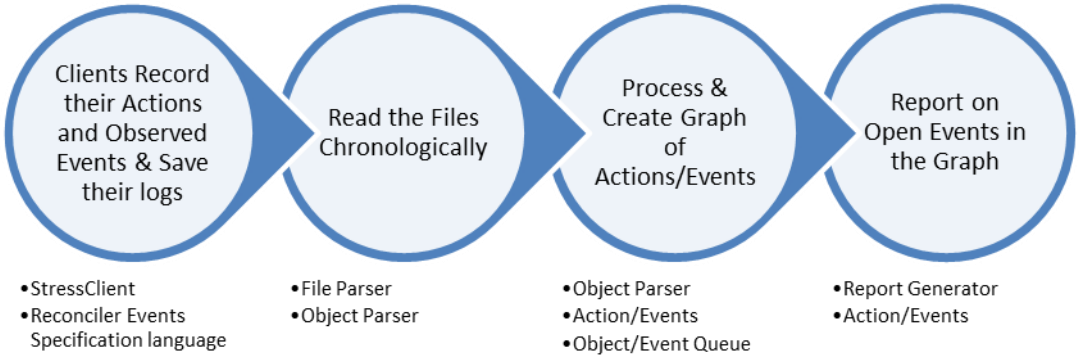
The Actions module is a layer that is built on top of existing test libraries. This module standardizes actions that can be executed by the AutoBugbash framework, allowing any type of endpoint to plug in and have the AutoBugbash framework execute its actions. The Actions module also keeps endpoint specific code separate from the randomizer component.
The Actions module contains actions for endpoints that extend the Action class. The Actions module follows the factory pattern, where it provides the specific Actions factory for a given endpoint type. Then the specific type of endpoint's action factory provides the specific action based on given parameters.
This Action class defines a method called Execute, which accepts an optional dictionary of parameters. The Action class extends each action type, supported by the particular type of endpoint. In the implementation of the Execute method, the applicable test library methods are called. The guideline for Action implementations is to do minimal state and event checking for each action. The state checking only checks whether the action can be invoked. There is also minimal event checking after the action is executed. This lessens the total time spent executing an action and allows more actions to be called in sequence. It also allows more code paths to be executed within the product code.

2.3 The Reconciler

One of the key tenets of the auto-bug bash is to do post-mortem analysis of expected functionality rather than real-time verification. This has several advantages:

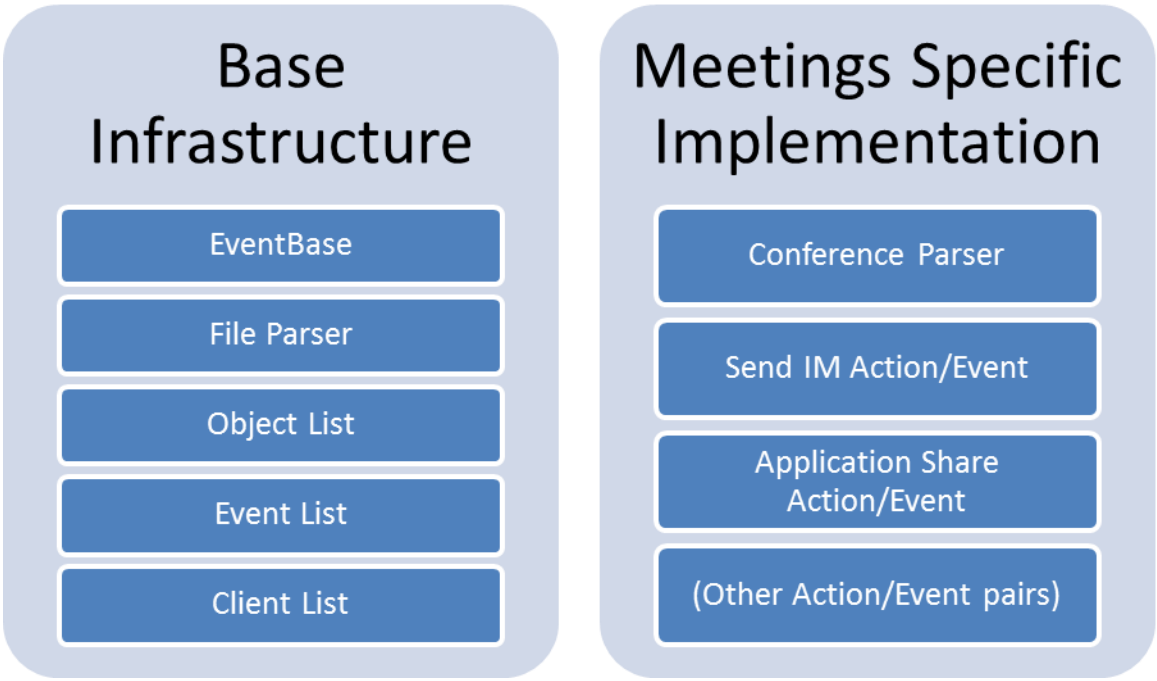
1. Ensures that verification does not slow down or change the scenario being tested – if 300+ clients were doing real-time verification when an action was taken, there would be significant slowdown in the execution.
2. Allows simple scalability – without a centralized component to coordinate actions, it is easy to remove or add clients at any time.
3. Allows multi-platform support – Doing post-mortem analysis allows there to be clients of various platforms joined to the meeting. A central component does not need to be able to speak to clients of multiple platforms (for example, mobile platforms, C/C++, C#, Silverlight, Java, etc.)

2.3.1 Reconciler Workflow



Clients will record their actions and events into a common format we call the **Reconciler Events Specification**. Regardless of the platform, a client can record its actions and events in this format for the Reconciler to parse.

2.3.2 Reconciler Components



2.3.3 Endpoint & Endpoint List

- Action Capabilities – lets the reconciler know which Action/Event pairs a client supports
- Client URI (Uniform Resource Identifier) – a string identifier identifying a client (such as user1@contoso.com).

2.3.4 Action/Event

The Reconciler breaks down the meeting logs into Actions & Events. An Action creates an expected Event in an "open" state. As the Reconciler continues to parse the log and encounters the Event, the open Event object is updated to reflect the clients that received that event.

- Event State
- Reference Count by participant URI
- EventBase class which provides basic matching infrastructure

Example:
The SendInstantMessage action creates an InstantMessageReceived event in an open state with each participant with a reference count at 1. The event is added to the event queue. As each participant receives the event, that participant's reference count goes to zero. When the last participant receives the event, all participants' reference counts are zero, and the event is marked as "closed."

2.3.5 Event Queue

The Event Queue is a list which keeps track of all created events. The reporting mechanism reviews all events and determines which events are still open.

2.3.6 Object Parsers

Object Parsers are parsers that understand a specific set of actions & events and act as factory methods for Event objects.

2.3.7 File Parser

The file parser reads all log files, processes them chronologically, and sends each corresponding line to the Object Parsers.

2.3.8 Meetings Specific Implementation

In the meetings-specific implementation, the Conference Parser object keeps track of Participants & Modalities in the conference. It creates Modality-specific events. Over time, the conference parser recreates the conferencing scenario. If an instant message is sent but not received by all capable endpoints, that event remains open. An open event is the indication of a functional failure; that is that all expected consequences of an action (in the form of events) did not occur as expected.
When the reporting processing is happening, all open messages are flagged as failures in the final report.

3. Rhythm of Execution

To get the most benefit from AutoBugbash, the best approach is to deploy the framework to a high number of distributed clients and have a long span of time for the run. Since many bugs are found on real user machines, we decided to deploy AutoBugbash on our team members' machines. To not interfere with their daily work, we provided a scheduled task to kick off an AutoBugbash client per machine. The run would start at 12am, and stop at 7am. This approach allowed us to use a pool of machines without disruption to the users.
We also automated the process of getting logs and dump files, in case of crashes, automatically. Selected members of the AutoBugbash team can look at dump files, logs and report from previous night's run from a central location and triage the issues.

3.1 Results

With the auto-bug bash, we hit several classes of issues:

1. Memory Leaks
2. Timing issues – Crashes, Incorrect Functionality, Incorrect state for the product
3. Reliability Issues -- Dropped Messages

3.1.1 Sequencing & Conditions

We did not hit many of these issues during functional tests. We found that running any small sequence of these steps in isolation would not reproduce the problem. But the exact conditions created by the constructed scenario would be different over time & expose different issues that only occurred under those conditions.
For example, switching sharers in an application-sharing scenario was a functional test case. However, switching sharers in conjunction with other actions, such as taking/granting control of an application-sharing session, created a condition that exposed a bug.

3.1.2 Environmental Factors

We also found that having AutoBugbash run on various people's machines gave us a large diversity of environments. Hardware specs varied greatly -- We had single, dual, and quad core CPUs, with various bits of memory. Operating Systems ranged from Windows XP through Win7, Windows 2k8 servers, and mobile/web runtime environments. Office versions would be different. All of these factors also exposed timing & interaction issues.

3.1.3 Time & Samples

Finally, running the AutoBugbash over time exposed new issues whether we used new daily builds or the same build day over day. This was the nature of timing issues – The longer the scenarios were executed, the more likely a timing issue was exposed.

4. Conclusion

The AutoBugbash project has been a great addition to our test portfolio. During the development of the AutoBugbash, we learned several key lessons:

1. Controlled Randomization – We achieved the best results through a controlled (weighted) randomization algorithm as opposed to using structured functional or completely random models.
2. Diversification of Environment – By running on machines used by engineers, we got a diversity of environmental factors such as installed & running programs, browser versions, desktop configurations, etc. that were not attainable in a practical way in a lab environment.
3. Frequent Test Execution – In a controlled randomized environment, frequent test execution yielded additional bugs as tests were running slightly different actions every time they were executed.
4. Multi-platform support made a difference – Having a post-processing verification system allowed for an easy point of convergence in a multi-platform environment. We did not have to invest in a test framework that would span across multiple platforms (mobile, desktop, PC vs. Mac, etc.)
5. Opt-in Model – As AutoBugbash is an opt-in model, we needed to quickly eliminate any and all barriers to adoption. For example, we found that Setup needs to be "one-click". We could not make any modification to the user's environment. As a "low-touch" solution, we had to provide auto-update functionality. Finally, we used a leaderboard to give recognition to those who participated.

How can others apply this model to their problem space?

There are several elements of our solution that can be applied to other problems spaces. In particular, this solution works well in the following situations:

1. Great for Multi-machine or distributed scenarios – no need to have a centralized controller or coordinator across multiple endpoints
2. Scalable model for multiplatform scenarios -- Decentralized model allows for incompatible platforms to work together

3. Great for Finite State Machine test environments where there are multiple branches in a particular state – rather than exhaust every possible branch, a weighted randomization allows you to hit a representative sample across several branches

What are the next steps for AutoBugbash's evolution?

Given our preliminary findings, we want to scale up the runs on several fronts. In terms of frequency, we would like to have tighter integration with our daily engineering processes. We aim to have an AutoBugbash run automatically on every build of the product. This scales up the frequency of the run. We want to scale up the test coverage by adding more daily users into the AutoBugbash. Finally, we want to scale up the tested functionality by continuing to add Action/Event pairs to ensure that Actions are implemented for all core functionality in the product.

Application Security for QA Managers – Pain or Gain

Dr. Ravi Kiran Raju Yerra
Arsin Corporation
ryerra@arsin.com

Abstract

It is often the case that developers and software vendors are not fully aware of application security vulnerabilities such as cross site scripting, injection flaws, cross site request forgery and etc. In many cases, these vulnerabilities can be prevented with training, more consistent and standardized software development practices, software acquisition protocols, and appropriate use of manual and/or automated security vulnerability test, manual and/or automated security code reviews.

Biography

Dr. Ravi Kiran Raju Yerra is an internationally known speaker with his doctorates degree in internet security management. Dr. Ravi Yerra has over fifteen years of real-world experience in delivering information security solutions around applications, cloud, virtualization, products, and database along with risk assessments and software testing across the globe in numerous industries and verticals. Since 1995, Dr. Yerra has been involved in multiple security projects and played a vital role in establishing various private and government information security initiatives.

1. Introduction

Historically, application developers and Quality Assurance (QA) teams have not focused extensively on security. IT management typically asks developers to achieve two goals - build innovative features and see that the project is completed on time. On the other hand, QA teams ensure that the application functions as intended and that it can scale effectively and perform under load (functional and performance testing). Unfortunately, during the development and QA phases management rarely ask that any real form of security testing take place. In fact, security testing is often viewed as an initiative that works in opposition to the aforementioned goals, as it can extend the already lengthy development and testing phases. Far too many organizations treat security as an afterthought as opposed to being integrated throughout the development process. In addition, most developers, QA professionals and QA managers do not consider themselves responsible for application security - assuming that security will be managed while the application is live.

1.1 Here are several myths of QA Managers:

Below are the most common security myths behind why QA managers don't always include security testing in the QA cycle in most modern and mature enterprises.

- Security Testing is already being done
- Security Testing is too hard
- Security Testing is not viewed as QA's job
- QA can't effectively perform security testing [they're not experts]
- QA analysts don't understand Security Testing

In this whitepaper we will discuss how QA teams can bridge the testing and application security gap in the software development lifecycle by debunking the several myths, actual reality and how a QA manager will benefit implementing Application Security.

2. The Problem - Application Security is a Quality issue

The majority of vulnerabilities in web applications reside in the custom business logic of the application itself. Compensating controls provided by external products are temporary solutions which seek to hide the vulnerability. It is typically only a matter of time before an attacker identifies an alternate entry point or is able to encode an attack in such a manner that a signature-based technology is unable to detect the attack packet. Only by implementing security during the Software Development Lifecycle (SDLC) is it possible to fully protect the application. This is the reason that developers, QA teams, and the QA management must share in the responsibility of implementing security at SDLC. Auditing a web application, either prior to or following release into production, simply is not sufficient to identify all vulnerabilities adequately. Application security is most effective as an iterative process that is applied consistently throughout the development lifecycle. The better way is providing security from the initial stage of software development i.e. security from the requirement stage to the deployment of any software product. This mainly involves the adaptations or augmentations of existing SDLC activities, practices, and checkpoints, and in a few instances it may also entail the addition of new activities and practices. Secure software is not easily achieved and the actual scenario is that investments in software development process improvement do not assure software that is resistant from attacks or modern security vulnerabilities. It can be demonstrated that changes to the software development process can help to minimize the number of vulnerabilities in new or developing software. Some of the risks posed by an insecure application are financial in nature and the cost of a single security breach can be significant. It is important to remember that the

total outcome can be difficult to fully measure due to the intangible nature of many costs. While the cost of labor to remediate the damage would be an obvious factor, damage to a corporate reputation caused by a defaced website or an unavailable application due to a distributed denial of service attack can be much more difficult to measure. Regulatory risk is another substantial and growing concern. Failure to adhere to a growing list of government and industry regulations can lead to fines, discontinuation of services, and even civil and criminal penalties. The common compliances all emphasize the need for security, especially at the application level.

2.1 Application Security - The Unfamiliar Limit for QA

Unfortunately, the availability of QA managers who have application-security testing knowledge is extremely limited. Existing tools such as static code analyzers or black box testing tools are complex and require security and vulnerability expertise that is rarely available within QA organizations. Businesses need a simplified, cost-effective means to incorporate security expertise into QA processes without impacting production schedules or resources.

In order to meet these rigorous requirements, QA managers must believe that security is important, security ensures quality, security is testable, and security makes a difference. It's crucial for QA managers to adopt security testing in a step by step process, including application security as a primary focus and eliminating any common excuses such as:

- We are behind a firewall.
- We use Secure Socket Layer (SSL) for authentication.
- We don't generate session ID.
- We don't use cookies.
- We use secure file transfer protocol (SFTP) to transfer data to a third party.

3. The Solution – Connecting Your QA Department

QA teams have some interesting ideas when it comes to answering this question: "Are you doing any application security testing currently? and depending on who you ask it's possible you will receive a variety of different answers. "Testing for security" can mean many things to many different people, and we wanted to take a moment to debunk some of the myths that spread over the last 3-5 years.

There are three ways that your Quality Assurance department may become involved with Web application security testing:

- Your company's Web security experts request that application security testing be completed by the QA group to ensure that all fixes have been implemented and no security holes exist prior to releasing the product to production.
- Your compliance officer, facing concerns about SOX, HIPAA, PCI, and so on, requests that further application security testing be performed during the QA process.
- Your QA department themselves request involvement with testing for Web application security, because an application that has security holes in it is not going to be perceived as high-quality by users.

No matter how the department gets involved, certain steps will need to be taken to establish the application security testing process. It will need to be determined whether there will be specific, dedicated staff members who will be performing Web application security testing, or whether the task will be dispersed throughout your entire QA group. In addition, the timing of Web application security testing during the Quality Assurance process will need to be managed. Ideally, application security testing will be performed as early as possible, so that developers can fix any security issues in a timely manner, without compromising the project's schedule. Finally, the right software for application security testing will need to be selected and implemented. Let us further explore the myths in detail

vs. how security is taking shape in the real world and learn the techniques how to integrate Application Security and enable QA teams to handle security issues. The below table debunks the previously talked about myths and guides QA managers about how to plan and integrate application security and benefit from it.

Myth	Fact	Tasks for the QA Teams
Myth 1 - Security Testing is already being done		
We are checking for the existence of the password requirement, or making sure pages aren't accessible without authentication doesn't actually mean you're doing security testing.	▪ Most security testing that I've observed QA organizations performing is centered on session management, and account authentication (and in some rare cases authorization). ▪ Very few QA organizations actually check for the types of issues that come from manipulating application logic, sending in garbage or malicious inputs, or attempting to break in to the application with malicious intent. ▪ Over the past five years we've seen an exceptional growth in the sophistication of application security measures. Unfortunately, that has been paralleled by an increase in the ability to penetrate applications. ▪ With recent attacks more targeted at Web applications, it has become increasingly important to address application vulnerabilities overlooked in the past. Though some of the companies are maturing and addressing application security issues, a vast majority either underemphasize the importance of those measures or labor under misconceptions.	▪ Identify a core QA Analysts and conduct a workshop on importance of application security. o Why Security & different trends? o Why Application Security? o What is vulnerability, threat, risk? o What is business risk? o Tools and techniques to identify and fix Security bugs. ▪ Setup readily available free to use IBM's Testfire and WebGoat, test environments for practicing application security skills ▪ Adhere to very well known web application security standards such as OWASP, WASC Threat Classification V2.0 in-order Increase security testing coverage. ▪ Apply a risk based approach ▪ Test with malicious intent.
Myth 2 - Security Testing is too hard [complicated]		
Security testing can be quite difficult, if there is no expressive and well-established framework or process utilizing tools and documented, repeatable methodology. *QA teams complain about security testing is that some security teams try to force a whole new suite of testing tools and methodologies on the QA*	▪ Any testing would be hard unless there is a pre-defined process and methodologies are available and practiced. ▪ In order to be successful it's critical to first understand the existing processes, tools and methodologies the QA teams use today and then adapt security testing protocols, tools and methods to fit within those existing processes and be	▪ Yes, security testing is too hard and complicated when there is no o Defined test procedures, guidelines, checklists, test cases & etc. o Refer OWASP Testing Guide for detailed methodology. o Train team on new set of tools that aid automating majority of application security testing ▪ Security needs to be tightly

Myth	Fact	Tasks for the QA Teams
analysts – that's a sure-fire way to meet opposition.	minimally invasive or disruptive. ▪ This issue is further intensified by the lack of proper process or guidelines. As a result, a vast majority of development managers rely heavily on pre-programmed tools such as IBM AppScan or WebInspect from HP to identify the vulnerabilities in their applications. Based on my experience in this field, I have found that the problem is two-fold: people rely on out-of-date information for their decision-making and they lack initiative for staying current in this rapidly developing field. ▪ With recent attacks targeting applications, firewalls and intrusion detection systems (IDS) are not sufficient.	integrated into the application, and there should be a proper secure design methodology. Managers, architects and developers need to increase their awareness and receive differential training. For example, developers should concentrate more on secure coding of their applications, whereas architects ought to focus on designing secure applications. ▪ The architect should look beyond using just SSL. Some points to consider while designing the module: o Is SSL being used only for login/authentication or for the entire site? o If the SSL is being used only for authentication, is it also used for the change password or forgot password page? o Does a session get created? If yes, does it use a secure cookie to store the session ID? o The document should also contain encryption detail such as key length, encryption algorithm, hashing algorithm, etc. Not to mention the fact that it should comply with the company's standards. o Where are the user ID and password stored? Are they stored in the database? Are they stored in clear text or encrypted? If encrypted, then is it one-way encryption? o If a user forgets the password, does a system send his original password in an e-mail or does the system generate a new unique one-time password and send it via e-mail? Is the user forced to change the password when he logs in for the first time? Is the e-mail that contains the password secure?

Myth	Fact	Tasks for the QA Teams
Myth 3 - Security Testing is not QA's job [that's security's job!]		
QA gets the application much earlier than security teams, for testing and finding defects earlier is always a bonus.	▪ Security is everyone's job, more explicitly – the QA organization is responsible for software quality – of which security is a component. Security is the overall pillar or quality to see does it function? Does it perform? Is it secure? QA teams own almost exclusively.	▪ Engage QA teams to provide training o Firstly a change in mindset o A bit of background on Web Applications o Basic things to integrate into your QA Process (Vulnerability Discovery) o A bit of validation ▪ Further Reading for moving away from basic to deeper understanding (Open Web Application Security Project (OWASP) & Web Application Security Consortium (WASC)) ▪ Confidently your team will be now prepared with some basic information on how to identify basic vulnerabilities within your web application as well as places you can search for more information on web application security while gaining more peace of mind. ▪
Myth 4 - QA can't be effective at security testing [they're not experts]		
A valid argument… if you completely disregard the fact that the application security testing tools market is maturing faster than our national debt. *QA testers don't instinctively think like hackers and work from functional specifications*	▪ Hacker look for landmarks a little different - but with the same principles as common is applicable to functional or manual testing. ▪ Shift your mind to a hacker touring your site or application for application security issues. ▪ Whereas if you look at they have quite opposite mentality difference: ⇒ QA asks - How does it perform? ⇒ Hacker asks - How can I break it? o To summarize one need his change his/her mindset - Think like a Hacker. ▪ Attackers look for "exploitable functionality whereas functional testers understand "use cases".	▪ The secret here is that QA analysts don't have to be security "hacking" experts if they have the right tools in place. QA analysts don't have to be expert hackers; in fact, organizations prefer them not to be. o Basic things to integrate into your QA Process (Vulnerability Discovery) o QA analysts just need to be able to use tools and follow process effectively to identify basic application security flaws. o Gain training from consulting teams just need to make sure that the tools and processes they set up for their QA analyst counterparts are templated and uncomplicated ▪ Quick hints for QA teams for finding hacker landmarks: o Look for changes in privilege or trust o Look for application interaction

Myth	Fact	Tasks for the QA Teams
		points o Look for opportunistic data validation o Follow the money (Commerce)
Myth 5 - QA analysts don't understand Security Testing		
	▪ QA analysts can provide needed support and enable security testing much earlier in the development lifecycle, but not without the proper training. ▪ It's the lack of exposure that's currently hindering adoption and fueling this final myth. o In reality, many QA analysts start to become very interested in hacking and security testing once they've been given the opportunity to learn and experience.	o Provide a foundation for understanding web application security testing by identifying potentially vulnerable targets to basic attack strategies and advanced tools based hacking techniques. o Guide them to examine the most common web application defects o Understand the concept of "negative testing" o Organize application security workshops on OWASP, WASC threats and vulnerabilities.

4. The Benefit - GAIN for QA Managers

Within the application development lifecycle, the QA Manager is responsible for both the resources and tools used to ensure that defects are identified early and do not materialize once the applications go into production be it Functional, Performance and Security issue. The cost of resolving a defect in production can be more expensive than capturing the defect early in the development lifecycle. The QA Manager will lead a team that will address the three pillars of quality:

- Does the application provide the functionality needed to meet business requirements?
- Does the application function with sufficient performance to meet business requirements?
- Finally and most importantly does the application deliver adequate security to meet business requirements?

Hopefully QA teams are now armed with a new approach to design Application Security Testing Practices within the QA group by integrating security across all stages of the software delivery lifecycle. You can search for more information on web application security while gaining more peace of mind. Here are few lists of direct benefits for QA Management:

- ✓ Equip QA teams to handle advanced application security testing skills
- ✓ Operate a modernize QA team that address the three pillars of quality i.e. Functionality, Performance and most importantly Security
- ✓ Lay down a roadmap to advance QA tester skills on application security
- ✓ Providing opportunity to enhance employee skills
- ✓ Increase test coverage by focusing on most important pillar of quality i.e. Security
- ✓ Bridge the testing and application security gap in SDLC by involving QA to identify and developers to fix security problems early in SDLC.
- ✓ Delivering adequate application security to meet business requirements
- ✓ Show value to customer by saving cost for resolving a defect in production which can be more expensive than capturing the defect early in the development lifecycle
- ✓ Introduce security from early stages of development and achieve software security
- ✓ Deliver high quality secure software and gain customer mileage

5. Conclusion:

Changing the paradigm

In order to break this cycle, we must change the way that we fundamentally approach application security. Gone are the days when anyone involved in application development can say “Security is not my responsibility.” Security is everyone’s responsibility as it has severe impact on the business if not taken seriously. We must integrate security throughout the SDLC, not just hastily add it to the end. This integration will only occur if we involve developers, QA teams, and the QA management in security. Making such a fundamental shift will not happen overnight, but it is essential if we are to stem the tide of applications riddled with security vulnerabilities which offer multiple attack vectors and leave enterprises wide open to attack. However, if security is not an integral part of your company’s development process for custom Web-based applications, your Web interfaces may be vulnerable. Ideally, security concerns should be addressed during development phase. Finding and fixing security defects early in the SDLC is up to 100x less expensive than doing so in production according to estimates by the IBM Systems Sciences Institute 2009 Cost per defect metrics.

Many organizations today are realizing the importance of bringing Security Testing under their QA umbrella and are doing so in a phased approach that includes enablement, test plan creation, and automation.

References:

http://www.testfire.com/ - Initially Watchfire Inc., published and hosted Testfire website having functionality of vulnerable demo banking services to detect web application vulnerabilities and website defects. Later Watchfire Inc. was acquired by IBM and all of its product suites operate under IBM Rational portfolio.

www.owasp.org/index.php/Category:OWASP_WebGoat_Project - WebGoat is a deliberately insecure J2EE web application maintained by OWASP designed to teach web application security lessons.

http://www.owasp.org - The Open Web Application Security Project (OWASP) is a 501c3 not-for-profit worldwide charitable organization focused on improving the security of application software.

http://projects.webappsec.org/w/page/13246978/Threat-Classification - The members of the Web Application Security Consortium have created this project to develop and promote industry standard terminology for describing these issues. Application developers, security professionals, software vendors, and compliance auditors will have the ability to access a consistent language and definitions for web security related issues.

http://h71028.www7.hp.com/ERC/cache/568162-0-0-0-121.html - This article is from Hewlett Packard that throws discusses how to incorporate web application security in to Quality Assurance process.

http://searchsoftwarequality.techtarget.com/tip/Application-security-Past-myths-present-excuses - Anurag Agarwal is the author of “Application Security: Past myths, present excuses” where he discusses the application security myths, excuses and solutions.

http://h30499.www3.hp.com/t5/Following-the-White-Rabbit-A/5-QA-Myths-Debunked-Why-QA-Doesn-t-Do-Security-Testing/ba-p/2407792 - Rafal Los is the author of this beautiful article that’s sheds light and clarified most common and important QA myths in very much detail.

https://www.owasp.org/index.php/OWASP_Testing_Guide_v3_Table_of_Contents - "OWASP Testing Guide", Version 3.0 - Released at the OWASP Summit 08.

RELIABILITY BEFORE YOU SHIP

Wayne Roseberry

wayner@microsoft.com

Principle SDET, Microsoft Corporation

ABSTRACT

Reliability is one of the most difficult areas to establish confidence before shipping the product. There is always that nagging question, "Will it be reliable enough for real world load and demand? Did what we build this time get better than what we had before?"

There are two parts to the solution:

1. Choose the right set of metrics and consistently measure against those metrics in every deployment. By carefully defining what you mean by availability and reliability metrics you can better assess if your product is headed toward success or miserable failure.

2. Build a toolset and methodology that joins the metrics to investigation and data collection

This paper will describe how the Microsoft SharePoint 2010 team used reliability and monitoring tools in lab and real-world environments to substantially improve service availability and performance. The presentation will discuss what our key definitions were for availability, failure and performance targets, and show how we used those to establish confidence in reliability before the product shipped

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), Micorosoft 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.

1 BACKGROUND

1.1 SHAREPOINT

Microsoft SharePoint Server is a web-based collaboration and content publishing application. Customers buy it to satisfy a broad range of business requirements that include the following:

- team communication and issue tracking,
- document management
- simple workflow processing
- enterprise search and information portals
- business application aggregation
- content management and publishing

SharePoint has been in the market since 2001 and has become a strategic part of the Microsoft product offering, with growth rates that outrun any server product in Microsoft's history. Sales of SharePoint to date have been largely enterprise oriented, strongly attached to Microsoft Office client sales. Microsoft likewise offers cloud-based hosting services for SharePoint.

1.2 PERFORMANCE AND RELIABILITY CHALLENGES IN 2007

SharePoint 2007 released with rapid success in the market. At six months, around the point of the first service pack delivery to channel, the SharePoint team started receiving troubling escalations from early adopters. The general complaint was regarding performance. Customer SharePoint deployments were either experiencing high latency rates, or were experiencing high volume of request failures. The anecdotal evidence of product performance issues were inconsistent with our own performance measurements.

After addressing a few customer deployments one at a time, we drew the following conclusions:

1. The lack of previous feedback from the market regarding performance problems were due to the long deployment and adoption cycles common with server products. Our early adopters, who were only just now getting past their server rollouts were an early indication of where the first real wave of customers would be at about 1 year from ship

2. We would not be able to handle the flood of issues by responding to them as maintenance requests one at a time via the hotfix channel. We would instead need to find as many key performance problems as we could by hunting for them and releasing a new drop to market, much like a typical release cycle.

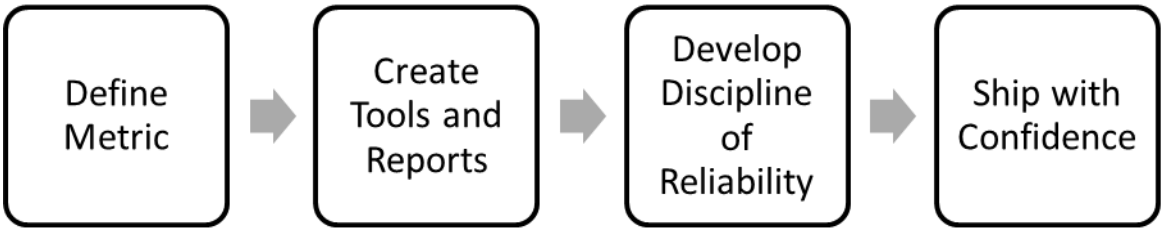
3. Our performance problems were actually reliability problems. SharePoint was fast enough delivering the typical request, but there were enough bugs in the product that would cause moments of high latency or server timeout to give the customer an unacceptably slow, or failed, experience.

4. Our previous means of assessing reliability were inadequate and we needed to fix them

This led to a large and expensive update of SharePoint 2007. The update was necessary to preserve product credibility and keep customers happy. The project deferred a lot of time and energy away from the SharePoint 2010 release. However, it was during this time that we learned many things about how to

approach performance and reliability engineering, which had a substantial impact on the SharePoint 2010 release.

2 WHAT WE AIMED TO ACHIEVE



2.1 DEFINE A METRIC AND MEANS TO TRACK RELIABILITY

In order to track and work toward progress, we needed something to measure. Discovering software flaws that would cause system failures and performance problems was not enough. We needed the measurement to determine the impact that those flaws were creating, as well as a means to motivate discovery of more problems. We needed to know how in-house production systems were actually doing so that we could assess on a regular basis if they were running more, or less reliably. The mechanisms in use by operations staff at the time were ad hoc. One means was to produce a visual graph of performance counter metrics for page delivery time, and then visually assess whether or not the data backing the graph had changed at all. This approach led to a great deal of inaccuracy, as the visual model was represented too densely, and the data points were not chosen well enough to draw any interpretation at all. We knew that we needed a quantitative, mathematical model based on well selected criteria. We knew that we needed tools to generate this model for us.

2.2 CREATE TOOLS TO IDENTIFY ROOT CAUSE

During the SharePoint 2007 release, we had a very difficult time linking evidence of failure (memory leak, server crash, growing latencies) with the root cause. There was almost no instrumentation in the product, and almost no tools available that assisted in the expensive task of examining log files for more information. We knew that the product itself needed more instrumentation, and we knew we needed a platform of tools for examining the data to find issues and draw correlations.

2.3 DEVELOP A DISCIPLINE FOR TRACKING RELIABILITY

Simply creating tools and reports does not solve the problem. We knew that we needed to make a regular practice out of reviewing metrics and discovering root cause for every instance of the product failing to meet goals. We knew we had to put together a team that saw chasing down reliability issues as their primary duty. We knew that we had to be able to account for every measurement that was off goal with a list of product flaws or problematic events which caused them.

2.4 SHIP WITH CONFIDENCE

Part of the problem with the SharePoint 2007 release was that we had no quantitative assessment of reliability. We didn't know what sorts of numbers we were getting, and we didn't know what any such numbers would have meant. This had caused us to ship not so much with a false sense of security as a vague feeling of uncertainty. That uncertainty manifested itself as a near disaster in market that was mitigated at a very high cost.

We knew that we needed to establish at least some measurement which quantified reliability in a way we understood. This would at least allow us to establish what we did know, and did not know about product reliability. That way, at ship, we would know at least if we were within the goals we were able to define.

The rest of this document describes how we met these goals.

3 DEFINE A METRIC FOR RELIABILITY

3.1 RELIABILITY BACKGROUND

The primary inspiration for our methodology came from Musa's (Musa, 1998). For those not familiar with Musa's approach, it can be summarized as follows:

1. Reliability goals are set based on customer perceived failure tolerances. The customer can tolerate failures up to a certain level. The product needs to meet that tolerance or better.

2. Reliability is generally either operation based, or time based. Examples of operations might be something like "percentage of jobs sent through the printer that print correctly without jam or failure" or "percentage of orders processed correctly without abort". Examples of time based might be "Hours available to service requests over total hours". The rule is that you have to pick which type of reliability you are going to measure, but you cannot mix both in the same metric.

3. To deal with the varying severity of failure, you group failures into classes of similar severity and weight them. Less important failures have less impact on the calculation than more important failures.

4. For values of reliability up to 95% there is a big complicated calculus formula. For values above 95%, the values drive toward a simple percentage calculation like "Attempted Operations/Total Operations" or "Available Time/Total Time".

There is a lot more to Musa's work than the above. He talks about how to weight operations in a load test, which failure levels to abort a run on, and how to use reliability test lab results to predict trends toward ship goals. For our sake, though, we were largely interested in how to build metrics around failure and reliability. To that end, we further simplified Musa's conclusions with our own:

1. Failure rates below 99% are so abysmally bad that accuracy on reliability becomes less important – so we were willing to tolerate simple percentages in our metrics, even for values of failure rate larger than 5%. We didn't consider ourselves even close to the ballpark until we crossed 99% (every 9 is an order of magnitude jump)

2. We had no idea how to weight our failures relative to each other. We didn't know if an "HTTP 500 server encountered an error" result was the same, more, or less severe than a page timeout. We skipped weighting different classes of failure and considered everything we captured equal

3. We realized that we wanted to know operational based reliability (# successful requests / # total requests) and time-based reliability (available time / total time). We also realized we wanted to reflect variance and reliability in request latency. We opted to measure all three, but as separate metrics.

The following section discusses how we defined and calculated those metrics.

3.2 OUR METRICS

As stated above, we chose three metrics to track, Failure Rate, Availability and Latency. All goals are stated as sustained in a real-world production environment for a period of 1 week.

3.2.1 FAILURE RATE: GOAL < .01%

Selecting a failure rate of .01% allowed us to target a success rate of 99.99%. We chose failure rate as an operations-based metric because single failures spread far enough apart have no impact on an availability calculation, but can still be frequent enough to cause substantial customer dissatisfaction.

1. Failure Rate: operations failed/operations attempted
2. We used the following definitions for failure:
    a. Timeout: Any request that takes longer than 15 seconds was considered a timeout and failed request
    b. HTTP error code: Any request returning an unexpected HTTP return code between 500 and 599 (the failure code range in the HTTP specification), 404 (not found), 403 (access denied)
    c. Error in body text: SharePoint pages will sometimes return with a successful HTTP response code (e.g. 200), but may have an error message in the body of the page. For these cases, we looked for simple indications of failure, such as the string "error" or "exception" in the HTML body of the response.

3.2.2 AVAILABILITY: GOAL 99.9%

While failure rate gave us a sense of how many requests were failing, most customers and systems operators talking about server and service reliability in terms of availability or uptime. We took the same data we were receiving to calculate the failure rate and used it to calculate an availability metric as follows:

    d. Availability: (Total Time – Downtime)/Total Time

    e. Downtime: We examined the timing of the requests for clusters of failures. If failures were packed close together in time, we counted that time block as being down. We defined a sliding window as follows:

        i. Window of time 10 seconds long

        ii. More than 10% failures occurring within a 10 second span defines a downtime window

3.2.3 LATENCY: GOAL 95TH PERCENTILE LESS THAN 500 MILLISECONDS

One lesson we had learned from customer issue in the previous release was that many of the reported performance problems manifested as intermittent moments of high latency. Most of the time the server

would deliver requests well below the time needed to render the page fully in 1 second or less, but periodically requests would start taking 10, 15, 20 seconds or more before even leaving the server. We decided we needed a reliability metric that was exclusively about page latencies, defined as time between the request being received from the client the time for the server to finish constructing the requested content and deliver it to the client.

We also knew that we could not take a simple aggregate of performance. Average latencies, even on systems where customers were reporting performance problems, were looking very good. The problem is that it takes very few outlier moments to create substantial customer problems. We decided we needed to target a distribution of latencies. We did this via the following method

1. Collect the time to last byte on client in milliseconds for a known set of pages
2. Sort all requests ascending by latency
3. Report the latency value at 25%, 50%, 75% and 95% of requests

By doing this we could tell whether the system was having performance problems with every request (at which point the values between 25-75% would be higher), or if the performance problems were isolated to outlier requests (at which point the values at the 95% would be higher). It also kept us from missing a performance problem that was hidden inside an overall average that was doing well.

3.3 THE MEASUREMENT METHODOLOGY

To collect these measurements, we had to build a few tools and make a few choices.

3.3.1 BUILD A RELIABILITY MONITOR

The first tool we built was an external monitor. This program executes a set of HTTP requests to the server. The program submits the requests repeatedly. It records the time between request submission and time to last byte. It records all error codes, timeouts, as well as any request that contains known failure messages in the content. This recorded is then aggregated into the reliability metrics stated above.

These metrics are collected and for every internal server that running SharePoint. We built a website that delivered daily updates on the metrics. The following is an excerpt from the report running against the Office team website, March of this year:

Reliability Metrics

	RTM	2011-03-31	2011-03-30
		Actual	Actual
Availability (%)	99.99	99.93	99.77
Failure Rate (%)	0.01	0.07	0.23
Latency: 25th Percentile (sec)	0.2	0.02	0.05
Latency: 50th Percentile (sec)	0.3	0.06	0.10
Latency: 75th Percentile (sec)	0.4	0.12	0.21
Latency: 95th Percentile (sec)	0.5	0.36	0.94
Avg APP Memory Used (GB)	6	0.00	0.00
Max APP Memory Used (GB)	8	0.00	0.00
Avg WFE Memory Used (GB)	6	0.00	0.00
Max WFE Memory Used (GB)	8	0.00	0.00
Time in GC counter (%)	5.0	1.06	0.94
WFE Crashes	0	0	9

3.3.2 PICKING THE OPERATIONS

3.3.2.1 WELL KNOWN PAGES

One of the first decisions we had to make was which operations to monitor. The problem with a rich, complicated server product that responds to user requests is defining a standard set of goals for all operations. We knew we wanted pages to fully render within 1 second on the client, leaving us about half a second (500 milliseconds) to finish processing the page on the server. We knew we didn't want to see failure code against user operations. Our debate was whether to base our measurement on a sample of requests or against all requests.

The argument for measuring all requests is simple. We didn't want to miss anything. This is a classic test dilemma, and can throw a test team into paralysis by analysis as you try to craft the test methodology which will catch all issues. The argument for measuring only certain requests is more complicated.

Not all operations are created equal. Some requests are incredibly fast, with delivery in single milliseconds, while some requests are known and expected by the end user to take a multiple seconds to

render. Our workload was dominated by lots and lots of small, fast requests, and we didn't want those to skew the distribution to the point of obscuring other requests. In the same regard, we didn't want to investigate latencies only to discover that higher latencies were acceptable for that particular request.

Even error conditions are difficult to equalize. Some errors should not happen ever (server crash, an exception thrown in the middle of a page body), but some errors are a legitimate response to an invalid end user request, such as "illegal operation", "forbidden operation", "invalid request" or "access denied". We knew we didn't want such errors for well-formed requests, but our measurement was not about determining how often the end user sent a bad request. Our measurement was about how reliable the server was under typical workloads – bad requests or not.

We opted to only measure a specific set of known requests. These requests were ones that should never violate the metrics we established – they should never return "access denied", or take longer than 500 milliseconds to render on the server. We decided that while this would miss failure on some operations, we would still be catching overall server health and reliability. We were betting that failure across the sample operations would highly correlate with failure on other operations.

3.3.2.2 PAGES CUT ACROSS RESOURCES, TIERS AND FEATURES

Once we knew we were going to work with a known set of operations, we knew we had to pick them carefully. We used the following criteria:

1. We would have a request representing each feature in the SharePoint product: Features in SharePoint are sometimes impacted by separate resources or components of the system, each capable of their own bottlenecks and failure points. Requests for documents from document libraries come out of a different SQL database than requests for a user's profile page. Requests to service end user search queries come from a completely different set of servers, files and databases than requests to deliver site pages.

2. The request would be one that was expected to be complete within 500 milliseconds or less: There are requests against database reports, or complex queries against large sets of data, which take a little bit longer. We decided to eliminate those to keep everything within the under 500 millisecond range.

3. The request would involve the complexity and cost of rendering a page body: Most pages viewed by end users have multiple HTTP requests, the bulk of which are tiny graphic object for rendering bullets, highlights and other page entities. Some requests for files are accompanied by frequent inexpensive polls on status checks. We decided to eliminate all of these smaller requests, particularly if they occurred in a background fashion where the end user might not notice them.

4. The request had to be relatively non-impactful on the system to execute: All requests put load on the system. We didn't want the measurement process to introduce a problem, so we intentionally picked requests that if delivered under a load necessary to get a good measurement would generally have little measurable difference to the system's reliability.

3.3.3 TUNING THE SAMPLES

Artificial load based monitoring systems require tuning. The problem is to balance accuracy of measurement against impacting the system. These two variables sometimes impose each other.

We wanted to be able to measure reliability all the way to 99.99% This meant that the sample intervals had to be as small as 1/10,000th of the entire time period. Any larger would make it impossible measure all the way to 99.99%. This works out mathematically as follows:

Seconds per day: 86400

Largest interval that could still measure to 99.99% accuracy: 8.6 seconds

In other words, out of an entire day, a single block of downtime lasting 8.6 seconds is the difference between 100% and 99.99%. We didn't want the occurrence of a single failed request to have that much impact. This meant we needed to send requests to the server at a higher frequency than once per 8.6 seconds, and that our failure window in the downtime calculation needed to be smaller than 8 seconds.

For request rate, we selected 2 requests per second. The product system at peak easily ran from 100-300 requests per second, so we felt fairly certain this would fit safely within our parameters.

We hit problems with these choices. One that came up right was search requests. For the initial production system, realistic requests per second from end users was running around 1 request every few seconds. At 2 requests per second, we were nearly quintupling the request rate above normal user load. To make matters worse, the search system cached recent results. The monitoring sample was forcing an artificial load pattern into the search cache, skewing distribution of data and pushing out real-world driven cache data, thus slowing down the end user experience for search users.We adjusted the search based requests to monitor at a lower rate, although by doing that we lost the ability to report on search reliability granularity to 99.99%.

4 PRODUCT CHANGES & TOOLS TO IDENTIFY ROOT CAUSE

Once we had actual measurements of production system reliability, we needed tools to aid in diagnosis of the causes of downtime, high latency spikes and operation failure. Some of these tools were built into SharePoint as features, others remained external. A brief listing of those feature changes are included below:

Unique Request Identification	Every end-user request was assigned a unique identifier that followed was recorded at every step inside the server via debug logs, error reports and database queries.
Detailed Debug Logging	A detailed logging system with varying levels of verbosity was tracked on ever server. In addition, a usage tracking system was introduced to allow identifying usage patterns and trends via a database query.
Monitored Code Blocks	The base class inside SharePoint was built such that it wrapped all requests with a special block that allowed for instrumentation and timing statistics. Data as captured in the debug logs, and any request that violated certain thresholds, such as length of time processing, was flagged as an exception in the debug log.
SQL Statistics Capture	SQL statistics, such as too much time spent executing a query, or queries that consume a great deal of IO and other resources on the database, were recorded inside the usage database.
Memory Usage Capture	Monitored code blocks tracked memory objects allocated and released. At end of the monitored code blocks the number of released objects did not match the number of allocated objects, an exception was recorded in the debug log.

Developer Dashboard	A special control was placed on all web pages which if enabled would deliver diagnostic information on the page containing time spent on each step of page rendering, all database and service calls executed on the page and a variety of other useful information to aid in performance diagnosis. The tool ships with the product and is available for customers as well (Microsoft, 2010).
SharePoint Diagnostic Tool	A special tool that aggregated usage database and debug log reports to help spot trends and isolate problematic requests. This started as an internal ad hoc tool and eventually shipped for customers to use as well (Microsoft, 2011).

5 HOW WE USED THE TOOLS

We had all the pieces of the above tools in place about halfway into the 2010 release cycle.

We introduced the tools into both test lab runs and real-world production systems. In the PNSQC 2010 conference I published a paper about testing simulated real-world load patterns in SharePoint 2010 (Roseberry, W.). We used the tools described in this paper to measure reliability in those test runs. This gave us parity with the production systems. We knew that we were measuring reliability on the same terms and definitions, the only difference being the load pattern. Any difference between lab results and production system results were either from the load pattern, the data set under load, or configuration of the system itself.

In fact, the simulated load-pattern tests were effectively finding performance and reliability bugs until approximately nine months before product release (around the time of the second beta release for Office 2010). At that point, load test results were almost always delivering 100% availability with very little evidence of latency problems, while production systems were returning approximately 90% availability and frequent spikes in end user latency. We made a conscious decision at that point to focus more of our time analyzing the production system failures.

We had the product in deployment in three internal systems:

- HTTP://Office The Office team departmental portal: this is where the Office product group shares information with each other, the rest of the company, and collaborates on projects and specifications

- HTTP://MSW The company-wide corporate portal. This site is used for searching across all content inside Microsoft's internal network, as well as finding key resources like legal, human resources, IT, facilities, etc.

- HTTP://SharePoint A company-wide hosting site for teams to have their own SharePoint site. Very similar to the http://office site, it mostly serves as a team collaboration and communication solution.

- HTTP://MY A company-wide personalization site, providing personal profile information for all employees in the company.

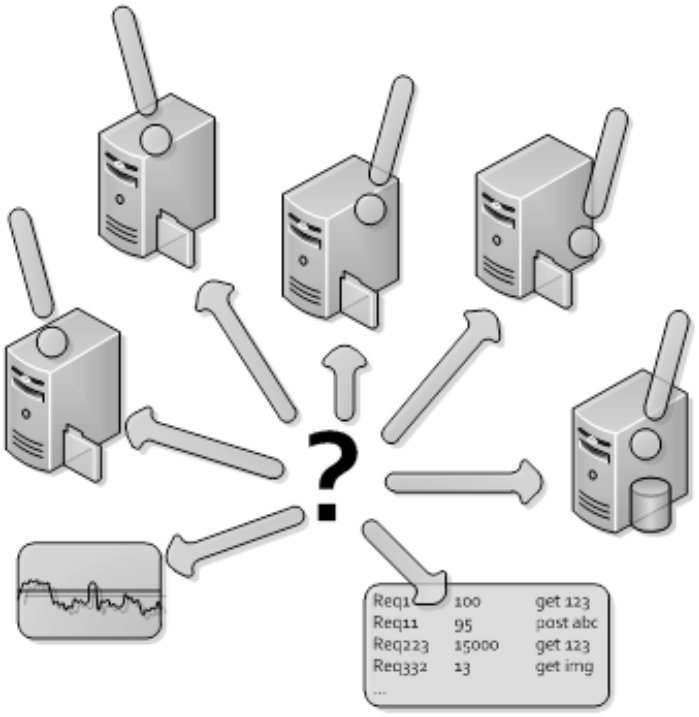
The hard work that remained was determining the best way to investigate reliability failures on these product systems when they happened.

6 INVESTIGATION

6.1 COMMIT THE RESOURCES AND BUILD THE TEAM

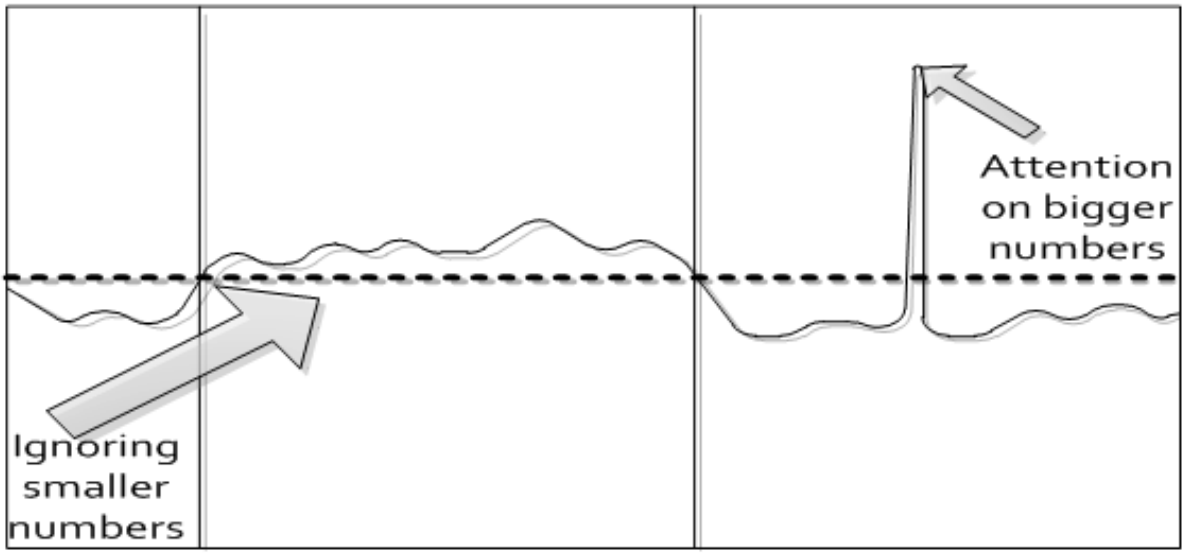
It is popular to say “Reliability is everybody’s job”, and while that it is true, it should not be mistaken as an argument against dedicating resources to specifically investigating performance and reliability problems. We found that properly addressing reliability a committed team of individuals actively investigating issues and hunting down root cause of problems. During the nine months of the SharePoint Server 2010 project, there were roughly a dozen or so people doing nothing but investigating performance issues on our internal deployments.

6.2 WHERE TO START?



Even with a set of tools at our disposal, our initial investigations were informal and random. The sheer volume of data in the system presented almost as much difficult to navigate as complete lack of information. Engineers tended to investigate problems based on what they were familiar with. If an engineer was really good at troubleshooting memory, then they spent a lot of time taking memory dumps. If an engineer had a lot of database experience, they would analyze query plans and disk utilization on the SQL server.

Another problem that occurred was how engineers chose which events in the reliability report to investigate. It was not uncommon for them to select the largest number available in terms of latency, mostly because such numbers stood out more readily in the report. The problem here was that there were sometimes hundreds of data points of this nature, and often the largest number was not the biggest problem.



The common aspect of the randomness was that engineers were examining failures at the individual level. Rather than look at the metrics defining availability and overall latency, they were looking to the lower level constructs that might indicate a problem. For example, a specific query on the database server might have been more expensive than others, so the engineer would begin examining that query to determine the reason why. While these led to many worthwhile discoveries, the process was slow, and often did not correlate to the most important problems that were showing up on the reliability measurements. Further, even when bugs were reported, it was not uncommon for the problem to be resolved as “Won’t Fix”, given that there was no evidence of how much of a problem the bug was causing.

We needed a methodology to guide engineers to properly investigate the most important problems.

6.3 FINDING PRIORITY IN STORMS AND SPIKES

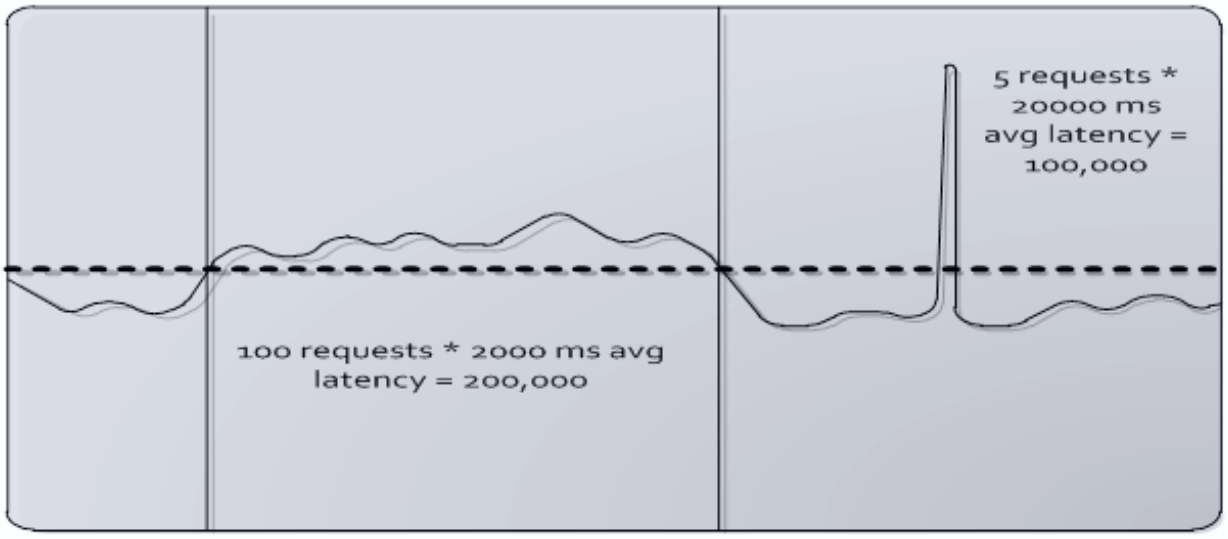
The solution to random investigation was to stop investigating individual requests and instead focus on periods in time. The idea was to treat reliability and performance problems as an event that had a weight, much like hurricanes are given a name and a weighting. This would aggregate multiple requests together into the same event, allowing us to prioritize our investigation by the event representing the biggest problem.

We created a construct called a storm. A storm was defined as a series of contiguous requests that failed to meet our criteria for success. A storm had two properties, average latency of the requests in the storm, and the number of requests in the storm. These were multiplied together to calculate what we called the storm intensity.

For example, imagine two blocks of requests, Storm 1 and Storm 2, each having the following properties:

Storm 1: 100 contiguous requests, average latency 2000 milliseconds

Storm 2: 5 contiguous requests, average latency 20000 milliseconds



Storm 1 has an intensity of 100 * 2000 = 200,000. Storm 2 has an intensity of 5 * 20000 = 100,000. While Storm2 has much longer requests, Storm1, with its longer time span of shorter requests actually accounts for an end user experience that is twice as bad. Both events were important, but Storm 1 rises to the top of the priority stack.

Even more important is the effect storm analysis has in regard to filtering out single requests that have little impact on overall reliability. Imagine that in the same time period there was a single request lasting 25000 milliseconds. While it is interesting, it is not nearly as important as the contiguous span of 5 requests in storm 2 all lasting 20,000 milliseconds.

6.4 INVESTIGATE EFFICIENTLY

The next technique we had to master was figuring out root cause of the problem as efficiently as possible. This is another place where the sheer volume of information overwhelmed us at first.

To put the volume in perspective, our production systems were typically running at 300-500 HTTP requests per second during the peak hour. Each of those requests would create around 1-2 dozen entries in the debug logs, and at least one entry in the usage log. The typical request would require 1 to 4 queries to the database, maybe more. This meant that even for a period of only five minutes of some sort of problem there were likely 3 million or more data points to examine for a possible cause. Some of these data points were difficult to investigate. An expensive database query, for example, may take days or hours of analysis before determining if it indeed contributed to the problem. With this many data points, and the cost of investigation being so high, we had to streamline our technique.

We came up with our approach by having all the engineers come together in a room, and say how they approached their investigations. Everybody laid their tricks out on the table, compared notes, and shared how they might do something differently. Doing this, we finally came up with a strategy that made investigation fast and efficient as possible, and more likely to find results that contributed to the problems customers were experiencing.

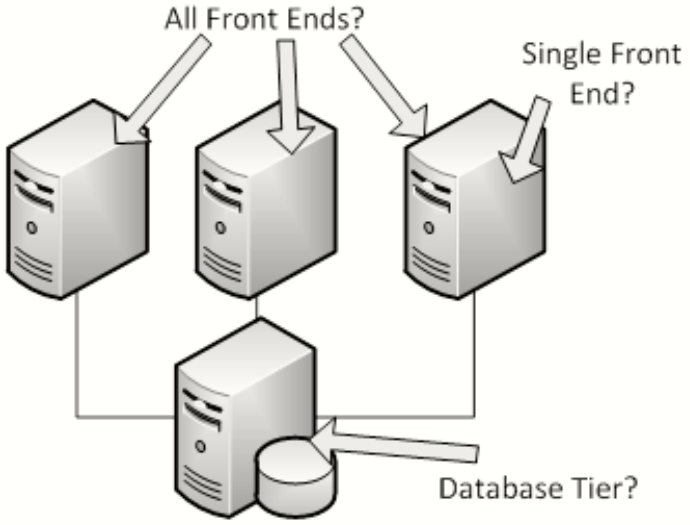
Here is the process we defined:

1. **Start from the reliability report numbers**: This guaranteed that the investigation would always be relevant to the metrics.

2. **Pick the time period to investigate based on storm analysis**: Pick the times which had the highest intensity. This guaranteed that the first things found would contribute the most to improving the numbers in the report.

3. **Isolate the requests that are within that timespan – focusing on the moment the problem started**: This allowed the number of requests to investigate to narrow even more, substantially reducing the amount of data required. Further, we noticed that whatever had caused the problem had to be happening at the moment the storm started. This allowed us to focus not just on the entire time span, but only those requests that were actively processing at the start of the time span, reducing the required requests from hundreds of thousands for a five minute storm to just a couple dozen.

4. **Isolate the tier having the problem:** This particular step is specific to systems like SharePoint, where components, services, and applications are distributed across many machines and resources. There were certain signals in the data, such as CPU performance counters, memory usage, length of time spent making external service and database requests, that allowed us to isolate very quickly whether a problem was occurring at a front end machine, at a backend service or database, or possibly across every machine in the system at once. This was so effective a means of reducing the investigation surface area that we sometimes went right to this step before even examining the request involved in the storm.



5. **Investigate the types of problem that occur at that tier specifically:** If we could see that all slow requests were coming from a single front end machine, then we knew that it was unlikely our problems would be in a shared resource, such as disk utilization on a shared database. This meant we would begin investigating CPU utilization on that machine, memory dumps, IO contention or serialized lock contention at the front end. If, however, we noticed that database queries were running long across all requests on all machines, then we would start looking at the database for expensive queries, IO intensive operations, disk queues, memory and CPU utilization.

6. **Correlate suspects to problem:** Once we had a list of possible causes, expensive database queries, serialized access for IO ports, memory consumptive operations, garbage collection, CPU heavy requests, we had to establish how well those requests correlated to the problems in the report. Were they the actual cause of the problem or not. To do this, we would look for other occurrences of the same problem and see if they lined up well with problems in the report. For example, if there was an expensive database query that occurred 100 times in the span of a day, and only two or three of those

7 SHIPPING WITH CONFIDENCE

Fortunately, the story here ends well.

First, we met all of our performance goals for our internal deployments. By the time we were shipping, all of the internal systems were running at < .01% failure rate, 99.9% availability and request latencies were easily less than 500 milliseconds at the 95% of requests. For the last two to three months of the cycle all availability incidences had been identified as caused by hardware, network or environmental issues outside the control of the product.

The mood and confidence at ship was a great deal different than with the previous release. We weren't naïve enough to believe that there would be no performance problems coming from the field, but we had solid numbers from production systems and we knew what they meant.

Even more important, though, was the response from support. We were monitoring the support requests, but we knew from previous releases that there would be a lag of at least six months or longer before there would be enough large scale deployments to really know if we had succeeded. We kept checking, and we kept hearing the same thing. Performance appeared to be okay. All evidence showed that our investment had paid off.

We knew we had a pattern and a methodology. We moved on to the next release, taking everything we learned with us. Indeed the methods, tools and learning we used shipping SharePoint Server 2010 would serve as the backbone of our next big investments in our hosted offerings in Office 365, running SharePoint in the cloud for millions of users worldwide.

8 CONCLUSION

The lessons learned by our team, and conveyed in this article, are not new. If anything, they were just the application of old principles and techniques. It is reassuring to learn, though, that taking a methodical, measured approach to establishing product reliability before ship is possible. The things you should do:

a. Pick your metrics and measure
b. Build tools, both external and inside the product, to make root cause investigation fast and efficient
c. Invest in time and people to investigate, using them to build better tools, methodology and find the root cause of problems

If you commit to similar kinds of approach to solving the problem as we did you should be able to ship confident that your product is meeting yours, and most importantly, your customers' expectations for reliability.

REFERENCES

Musa, J. *Software Reliability Engineering*. Osborne/McGraw-Hill

Roseberry, W. *"Simulating Real-world Load Patterns When Playback Just won't Cut It".* Paper presented at the Pacific Northwest Software Quality Conference, Portland, Oregon, October 2010.

Microsoft. "SharePoint Diagnostic Studio 2010 (SPDiag 3.0) (SharePoint Server 2010)." Last modified April 28, 2011. http://technet.microsoft.com/en-us/library/hh144782.aspx.

Microsoft. "Using the Developer Dashboard." Last modified May 2010. http://msdn.microsoft.com/en-us/library/ff512745.aspx.

Creating a Lean, Mean Client Integration Machine

Aaron Akzin: Quality Assurance Analyst; Shelley Blouin: Manager of Software Development; Kenny Tran: Integrations Developer; Supriya Joshi: Quality Assurance Analyst; Sudha Sudunagunta: Quality Assurance Analyst; Aaron Medina: Quality Assurance Analyst

Abstract

Reorganizations are all too common in the modern business environment and in particular at technology organizations. This paper chronicles the motivation for undertaking such an effort. We are all aware of the disruption this will interject upon employees, clients and business objectives and the inherit risk in doing so. Despite this, often these actions are necessary. The intent of the authors to you, the reader, is that we detail what was done, how it was done and finally a retrospective about what was successful and what requires additional improvement efforts. WebMD Health Services is a rapidly changing and dynamic organization and we refuse to remain static and dormant which we adamantly oppose in our culture and as a trait in the employees we hire.

Biography

The authors of this Paper are all members of a team within the Integrations department that are under the umbrella of Technology at WebMD Health Services. We all serve the Health Care Market for our customers that are in this vertical.
Shelley Blouin is a Manager of Software Development.
Kenny Tran is an Integrations Developer on the team.
Supriya Joshi, Aaron Akzin, Sudha Sudunagunta and Aaron Medina are Quality Assurance Analysts on the team.

Collectively we have decades of experience in technology in all roles and enjoy our interactions in this new structure and are extremely interested in sharing our experience during the successful transition.

CREATING A LEAN, MEAN CLIENT INTEGRATION MACHINE

1. Introduction

In late 2009, WebMD Health Services (WHS) started a process to undergo a radical transformation in its organizational structure. This paper will be a retrospective of not only what went right and what went wrong but also the challenges teams faced during the transition towards a collaborative and cohesive environment.

Before the change, WebMD's client integrations division consisted of segregated development and QA teams. Minimum collaboration and lack of mutual understanding between the two teams resulted in duplication of efforts, finding defects late in the project cycle and delays in delivering enhancements on time to our clients. Realizing a need for increasing client satisfaction and delivering higher quality features within shortened timeframes, WHS decided to take a leaner approach and create a more integrated environment by adopting some of the more common Agile principles and molding them to support our processes and tight client scheduling limitations. While we tried many different techniques, the following are the principles that provided the most benefit to our client-driven development cycle:

- Shortening duration of job assignments
- Reducing the amount of work in the queue
- Creating small, combined and cross-functional teams of development and QA to allow for greater flexibility and better coverage depending upon priorities
- Introducing daily stand-ups and weekly sprints that were timed with deployment schedules
- Prioritizing tasks
- Introducing the role of Market Champions to facilitate Sprint planning and communication with client-facing staff

Throughout this transformation, a real sense of an often cited cliché of "Teamwork" has been enjoyed and evidence is abundant of how this has ultimately resulted in better products and client satisfaction both internal and external. This is often demonstrated on frequent occasions as the definitive delineation between developers and QA has been "blurred" as they pick up each other's tasks as needed in order to meet deadlines and lend expertise as needed.

2. About WHS

WHS is a consumer health solutions company that helps people make better health and benefits decisions, positively change their health behavior, and live healthier lives. We have a customized, integrated set of health management and consumer-guidance resources. Our online health portal is personalized for each individual and branded for their organization. The platform integrates content, tools, and programs from WebMD along with those that are specific to their population to help you drive consumer engagement, health, productivity, and satisfaction (Overview, 2011).

3. About The WHS Client Integrations Team

The client integrations teams collaborate with the client services teams to create online health portals that meet client needs. Clients may select any combination of tools and solutions we offer.

This story is about the journey of WHS from waterfall to an Agile(ish) Software Development Lifecycle (ASDL), responding to change when needed and finding ways to improve client satisfaction every single day (Fowler, 2005). We do not subscribe to any formal ASDL method. So this story is not about applying Scrum or Extreme Programming. It is about taking a leaner approach to software development lifecycle.

4. The way we were

Before March of 2009, WHS development teams followed a typical industry standard waterfall software development life cycle and we all fit nicely in either the development (Dev) or quality assurance (QA) teams. Some of the challenges we faced following a waterfall ASDL were:

- As is common, developers outnumbered quality assurance (QA) by approximately 3 to 1, yet often the verification tasks took longer to complete than the corresponding development tasks. When a project was late, QA stayed late.
- When a request came in for work, a project manager approved it. It became one of many in an endless personal queue for individual developer and QA contributors. There was an infinite array of "tracker" (ticket) items to complete that developers and QA worked on separately. They set their own priority to complete, as long as the production date was met. This meant that QA might be calling developers at home at 9:00 PM to fix a bug that had to go the next day.
- Each of our clients was assigned one developer and one QA in an effort to ensure that we had experts for each client. However, there were challenges with this approach if you ever wanted to go on vacation. Developer and QA were responsible for vacation coverage as part of their vacation planning.
- Communication happened through email and tracker items. Collaborating on a project meant creating a defect tracker for every issue found or questions that needed to be answered. These items would not only burn cycles for developers and QA staff, but also administrative personnel such as Project Managers and Managers that would need to approve, schedule, and prioritize every item that was submitted.
- Most challenging was the long list of rules of engagement between the two teams. It was dogma that developers did not test and QA would conversely never touch the development environment.
- QA and testing starts only when development is complete.

5. What prompted a change?

At the end of 2009 our technology leadership changed. With this change came a proposal to take a more Agile approach to our software development life cycle. While our product development teams were eager to take on this challenge, those of us responding to daily client requests were terrified. How could we possibly adopt principles of the ASDL which, according to Beck et al. (2001) are:

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. (Well, we are already trying to do this…)
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage. (Given our tight timeframes and commitments, changing requirements are our worst nightmare!)
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. (Hmmm, we already deliver software daily – so this one might be easy.)
- Business people and developers must work together daily throughout the project. (This one, we really needed to work on.)

- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. (We have motivated individuals!)
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. (Lots of work to do here…)
- Working software is the primary measure of progress. (Yes!)
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity--the art of maximizing the amount of work not done--is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly. (Post-mortems, again?)

While we already released software daily, shortening our project time frames, ending individual Dev and QA client ownership, and getting rid of our project managers seemed like a daunting task.
Over the first six months of 2010, we created smaller, cross-functional, market-focused teams and took a leaner approach to developing software. By no means would you call us an Agile team of the purest of sorts, we have taken a few of the Agile principles and used them to create an environment that works for our professional services, client-focused environment.
The following changes are what have helped us create a leaner approach to software delivery for our customers.

People

- Creating small, combined and cross-functional teams of development and QA to allow for greater flexibility and better coverage depending upon priorities
- Introducing daily stand-ups and weekly sprints that were timed with deployment schedules
- Inviting client services folks to review our work early, even on test!
- Collaborating with technical business analysts and functional business analysts in order to get requirements right the first time

Work

- Introducing the role of Market Champion (MC) to facilitate sprint planning and communication with client-facing staff.
- Prioritizing tasks
- Shortening duration of job assignments
- Reducing the amount of work in the queue
- Holding retrospectives for larger projects and documenting our ideas for doing things better next time

6. How the Change was Implemented

Create small highly functioning teams

The first step towards a leaner approach was to dissolve the larger development, QA and design groups and create smaller (5-7 team members) cross-functional teams to support our three client markets: Employer, Health Plan and Distributor. We sized these teams based on the previous year's data. The Employer and Heath Plan markets each have two teams, while the Distributor market has one. This organizational structure in the Technology Client Integrations mirrors the structure in the Client Services organization.

Par for the course, we picked fun team names (Cerberus, Hydra, Kraken, Phoenix and spent the first few months trying to build strong teams and culture. We participated in team building events such as Tree–to–Tree (a ropes course in the trees!), had off-site parties, went to Taco Tuesday at specific team members houses, designed team shirts, and invested in team notebooks – anything to build a strong team identity.

It was interesting to watch how the market teams grew and positioned themselves to prepare for the Open Enrollment busy season. The Employer teams Cerberus and Phoenix grew into strong, distinct teams that did not share work across teams. The Health Plan teams Kraken and Titan eventually decided they would prefer to be known as Thundercats and work together as one larger entity and commit to work weekly as a team.

Manage the Work - Introducing the Market Champion!

Unlike product development, professional services work does not really lend itself to having a product owner because we are doing client work, not product development work. However, in order to function in a somewhat agile manner, we needed someone to prioritize and manage our queue. Before our organizational change in technology, there really was not much prioritization done…all work was approved and all work just magically got done (whether at midnight or six a.m.).

We introduced the role of the Market Champions (MC) in client services. MCs now approve, prioritize and manage our backlog queue of work for us and the team is able to commit to a certain number of points each sprint that we think we can do. We plan each sprint about 10 days in advance. Knowing that we will have expedited client requests, we reserve a number of points each week for expedited and urgent requests, based on historical data.

Our weeks look something like this, based on a weekly sprint with a release day on Wednesday:

Sprint	Monday	Tuesday	Wednesday	Thursday	Friday
Sprint 1	Sprint 1 work	Sprint1 work	Release Sprint 1	Start Sprint 2	Sprint 2 work
Sprint 2	Sprint 2 work	Sprint 2 work	Release Sprint 2	Start Sprint 3	Sprint 3 work
Sprint 3	Sprint 3 work	Sprint 3 work	Release Sprint 3	Start Sprint 4	Sprint 4 work

Managing Planned Work

Thursday is our big planning day. Here is how we make it all happen.

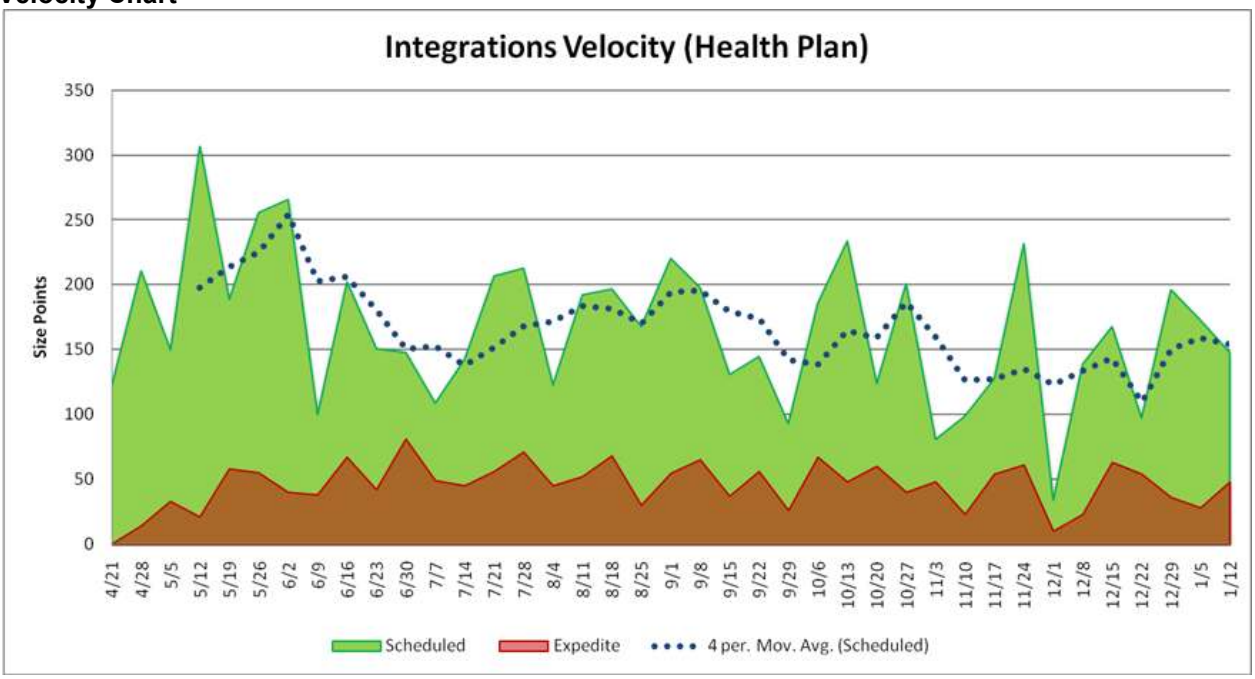
First, the team meets with the MC to review all of the work in the queue and size it as a team. When we started out, we used all of the poker planning card numbers to size our work. After a few weeks of haggling over .5s and 2s, the team opted to use a subset of the numbers for most of our planning:

1 = tiny – we can do a few of these in a day
3 = small – we can do two of these in a day
8 = medium – we can do this in a day
20 = large – we can do this in a week
100 = huge! – we will have to break this down and work on it over multiple sprints

Occasionally we have the need for a 5 or 13 when and 8 or a 20 isn't quite right, but we mostly stick to the sizing numbers above.

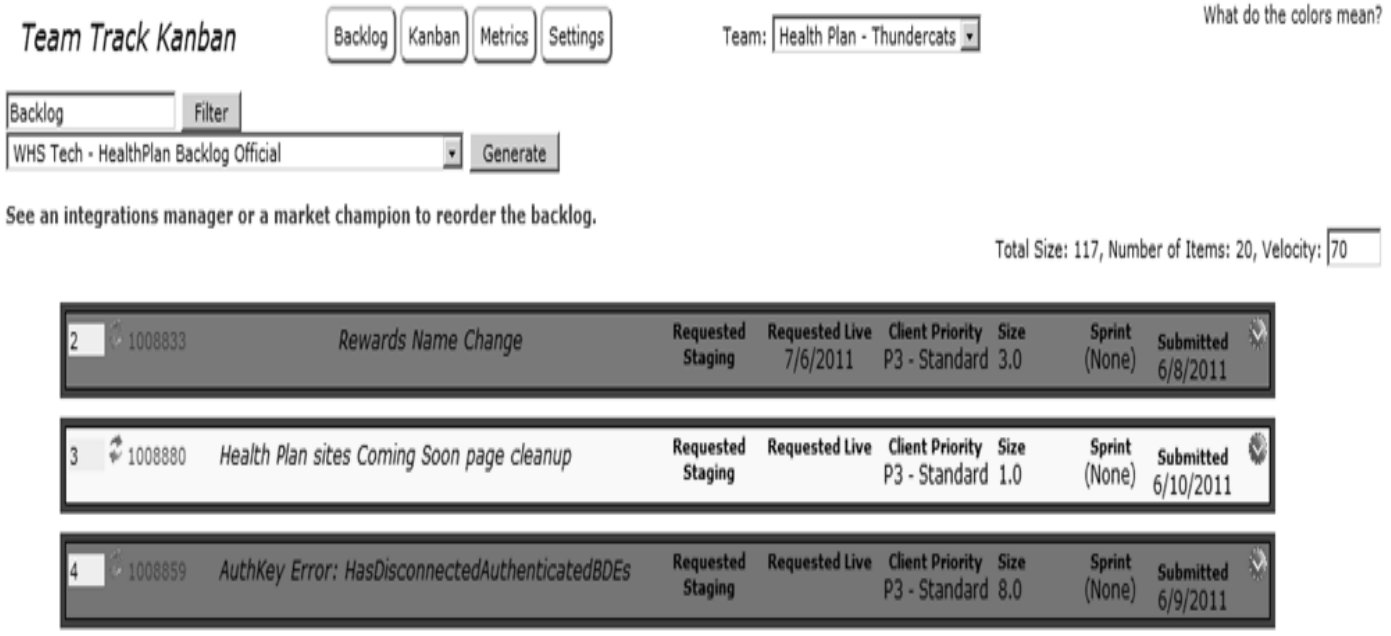
Second, the MC and the team managers determine what our velocity should be for the week. Tracking over several months has shown that we average around 150 points of velocity per sprint, which is about 15 points per person for Thundercats.

Velocity Chart

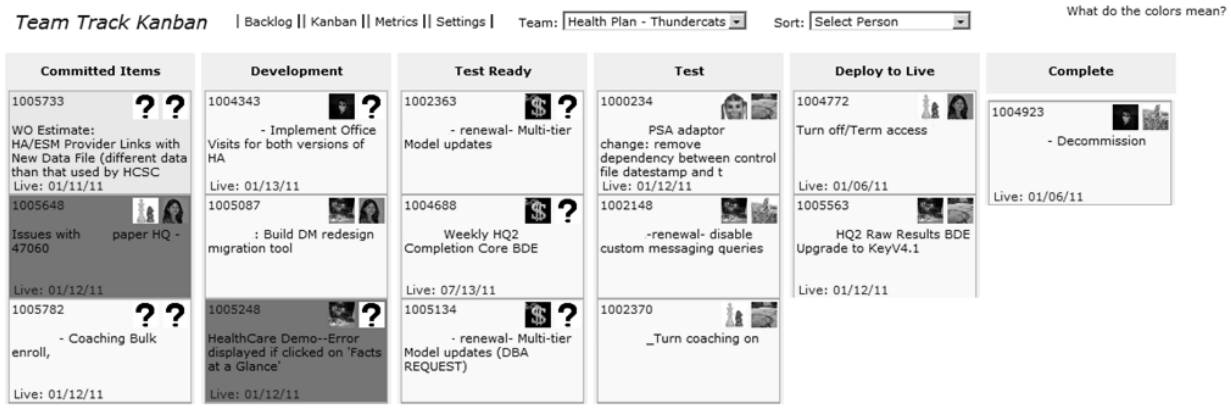


We've developed an in-house electronic Kanban Board and backlog tool that the MC can use to quickly determine how many items we can take for a sprint once we've determine velocity. The team usually commits to doing 100 points of work per sprint and reserves 50 points for expedited or urgent items that cannot wait until the next sprint to be worked on.

Backlog Tool



Kanban Board



Third, the MC and team managers review the list of items we believe we can commit to for the sprint with all of the stake holders in Client Services. This list has already been prioritized by the MC and most often is an accurate representation of what should be done first. However, this weekly meeting is a forum for making trade-offs when we do not have enough velocity to cover the work that needs to be done. It also facilitates conversations about moving work to other integrations development teams if necessary to meet client needs.

Lastly, on Monday, the team manager sends out a list of committed items that will either be worked on or completed for the following sprint that starts on Thursday of that week.

Committed Work

Item Id	Title	Staging Date	Production Date
1007969	Claims DI schedule request		4/28/2011
1007957	alter text on WebMD id card		5/4/2011
1007912	site manager issue		5/4/2011
1007953	Site Move from staging to production		5/4/2011
1007995	Turn on Challenges slide	5/2/2011	5/16/2011
1007946	PSA results page getting cut off on the right when emailing from add list		4/19/2011
1007939	Redesign proposal: Spec #2 (research)		5/4/2011
1007948	Privacy Policy	4/27/2011	5/4/2011
1007949	Privacy Policy	4/27/2011	5/4/2011
1007962	WO Estimate: create custom hydration tracker		4/29/2011
1007907	BDE notifications		5/4/2011
1007996	Add 3 new nodes to support coaching implementation	5/11/2011	5/30/2011

Managing Unplanned Work

Teams have daily stand-ups. We each do a quick review of what was completed the day before, what we are working on today, what we need help with and whether or not we can help someone else. Once we make the rounds with the team, the MC reviews any expedited requests that have come in since the previous day. We review and size them as a team and the MC puts them into our backlog as committed work.

The MC and managers keep a close watch on how many expedited points are being used per sprint. As we approach capacity, MCs and managers will work with the client services teams to either make trade-offs or find help from other teams to get the work done.

7. What has gone well

Improved client satisfaction

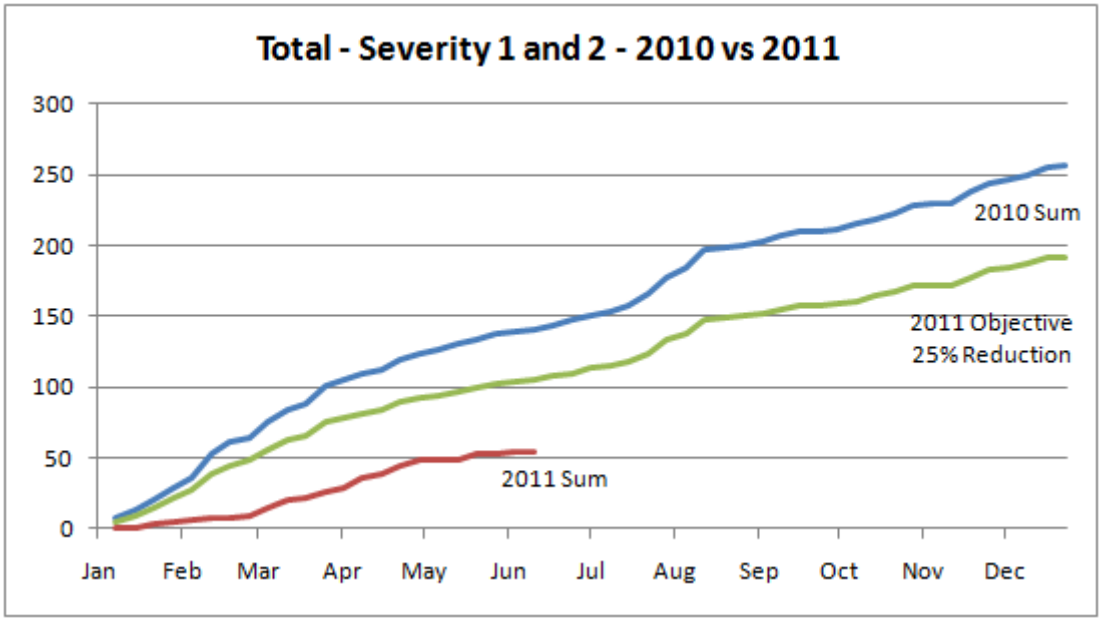
Our Health Plan clients have a whole team to support them instead of two individuals. We all know our clients well and anyone can take on any request. We also have developed close relationships with our client services team since it's a limited number of people. Before we aligned by market, each Dev or QA could potentially have daily interactions with every client advisor in the organization!

Improved Cycle Time

Projects that used to take at least two weeks to complete, are now completed in one sprint (week). Only items that are larger than a 20 should take more than one sprint to accomplish. New client implementations used to require 3 weeks of Dev time, 3 weeks of QA time, and then 3 weeks of staging time. Working together as a team and with the client services group, we've been able to reduce the development and QA time to less than 3 weeks because both Dev and QA can work on any task needed to get the job done. Anyone can test; anyone can code. We even created a new client site, including a web service and SSO in just one week!

Fewer defects on production

As part of the transition to a leaner approach WHS set corporate objectives reduce the number of defects found on production by 25%. Our Severity Committee has determined that we met this goal for 2010, which we believe to be a direct result of our changes.



Increased Job Satisfaction

Team members know that they can go on vacation. They know it's ok to ask for help to complete a task. Rarely does anyone on the team have to stay late because the whole team is focused on completing the work. QA folks are learning development tasks and leading testing efforts across the team, including developers.

8. What we still need to work on

Despite best efforts, our teams were not right-sized for our markets. Our employer market sees the most project work and requires the most resources. Because the employer teams were engaged in large projects, it was difficult for them to manage any maintenance or expedited work that came in. Last year, this maintenance work spilled over into different teams, which was difficult for a few reasons:

- Non-employer market teams didn't have the background knowledge to work on defects or research items that came in.
- Non-employer market teams tended to get the most unappealing, yet necessary tasks, such as work order estimates, to keep things moving along.

9. CONCLUSION

In summary, WHS determined it had to take bold and timely actions in order to increase throughput, productivity, client and employee satisfaction. We did this by reorganizing our Technology Client Integrations department in an Agile manner and staffing according to defined markets instead of client assignments. After our transformation it is a radically different culture which has produced immense benefits in all the areas previously identified.

Over the course of our transition we tried several approaches some of which worked where as some did not. Overall, the Transition was a success. Dev and QA teams are working collaboratively to deliver features in a timely manner. The rate of production defects has gone down and client satisfaction has increased. Employee satisfaction also increased as a result of improved work-life balance.

There are still some things we need to work on like sizing the teams based on workload. We brought members of other teams into the Employer team and split this into two groups. So far this has been going well and as expected. We are continuing the journey because we know we are on the right path.
We would like to close with some comments from our Client Advisors and their Director about how this transition proved a success for them too…..

"Years ago when client services and integrations were at odds with each other because there was no process and a huge gap, it made impossible situation for everyone due to tension and conflict." But now that there is a process that we follow, we have conversations involving both the sides and have market champion to make sure no one drops the ball makes it easy to understand the expectations. Communication helps! Now we are not afraid of speaking up because we know that our voice is being heard.

"Direct visibility into velocity and sprints lets us better manage client expectations and understand capacity."

"Always people first - has drastically changed relationships."

"Workload / client satisfaction side is far more aligned than what it used to be."

"Overall my clients have noticed that we complete work in a more timely manner and are able to maintain schedules. This has been a much appreciated change in our service level."

References

Beck, K., et al. (2001). Principles behind the Agile Manifesto. *Manifesto for Agile Software Development*. Retrieved from http://agilemanifesto.org/principles.html

Fowler, M. (2005). *The new methodology*. Retrieved from http://martinfowler.com/articles/newMethodology.html

Overview (2011). WebMD Health Services. Retrieved from http://www.webmdhealthservices.com/about-us/

Glossary

Kanban Board: defined by Miller at http://geekswithblogs.net as "essentially a signal, and can be duplicated by using a whiteboard, index cards on a bulletin board, sticky notes, or more elaborate means".

Market Champions (MC): approve, prioritize and manage our backlog queue of work.

Sprint: a time period in which the team does whatever it can to implement features/requirements that the Team has committed to.

Velocity: a measure of productivity sometimes used in Agile software development. The velocity is calculated by counting the number of units of work completed in a certain interval, determined at the start of the project.

Design for Delight Applied to Software Process Improvement

John Ruberto
John_Ruberto@intuit.com

Abstract

New product designers use a variety of techniques, blending art and science, to design the latest gadgets. Laptops, cell phones, kitchen utensils, and automobile dashboards are examples of products that have benefitted from the design process. My company uses a methodology, called Design for Delight, to create new services and offerings for our customers.

These creative methods also work for software process improvement. This paper shows how our team applied Design for Delight (D4D) for software process improvement. The paper will provide an overview of Design for Delight, tell the story how we applied it to improve a key software process, and describe the benefits and limitations of using Design for Delight for process improvement.

Our experience shows that D4D works well to identify customers of a process, work with them to learn the pain points, and identify improvements that focus on solving these pain points. The team is engaged and brings lots of creativity to problem solving.

On the other hand, we experienced several limitations of using this methodology for Software Process Improvement; for example, the results were highly dependent on the individuals selected as representative customers.

With these limitations in mind, our experience is that using product design techniques to improve software processes is a useful practice for applying creativity and innovation in software process improvement.

Biography

John Ruberto has been developing software in a variety of roles for 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.

1. Introduction

Intuit developed Design for Delight (Shortened to D4D) to design easy to use products that delight our customers, and it is cited as the reason for many of our recent product innovations (Martin 2011). D4D is a methodology for understanding customer problems, and finding creative solutions to those problems.

Our quality team was evaluating several process improvement methodologies, when the thought came to apply D4D to software process improvement. This paper describes D4D, and our experience using D4D for process improvement.

2. Continuous Improvement and Delivering Quality

Excellent organizations, the market leaders, largely get that way by constantly learning and improving. They quickly and relentlessly solve problems that impede progress, and move past their competitors in the process (Spear 2009).

Organizations use many process improvement frameworks to drive continuous improvement; among these are Lean, Agile, Six-Sigma, Total Quality Management, Reengineering, and the Toyota Production System. But, each of these frameworks essentially boil down to 3 main elements (Spear 2011):

- Define the process to capture the best-known approach
- Measure and monitor the process
- Solve observed problems with discipline

Spear implies that it matters less which process improvement framework is applied, as long as it's applied relentlessly. With that in mind, our team decided to use a set of improvement tools created from another discipline: product design.

This principle of excellence through continuous improvement applies to quality and software development organizations as well. Continuous improvement is an explicit component for agile process like Scrum – with the focus on retrospectives and learning (Schwaber 2004).

3. The Creative Process

Product design and creativity is a business now, with defined processes and schools that teach the art of design. Creative designers follow a methodology to manage the creative process. These designers are not necessarily experts in the field in which they are innovating, but are experts in the process of innovation. The creative process is a blend of art and science (ABC Nightline 1999).

The creative process essentially follows this framework:

- Study the problem – decomposing the problem into components
- Brainstorm many solutions, using people with different backgrounds.
- Evaluate those solutions, and chose a few to test
- Test solutions with prototypes and storyboards.
- Iterate and Refine until satisfied with results.

The appendix provides several references to specific design techniques.

Our team decided to try to apply this creative process to software process improvement. We used the Design for Delight process, developed for and by product designers for Intuit, Inc. (Martin 2011).

4. Design for Delight

4.1 Introduction to Design for Delight

Design for Delight is the product design methodology created and in use at Intuit, inc. (Martin 2011). The D4D process has three main components:

- Deep Customer Empathy
- Go Broad to Go Narrow
- Rapid Experimentation with Customers

The following sections provide a brief overview of the D4D process. The section after that describes in more detail how we applied D4D to process improvement in our organization.

4.2 D4D Roles

4.2.1 Customers

One tenet of D4D is to interact directly with customers, actual customers who have the problem. This allows the designers to hear raw, unedited and unfiltered, descriptions of the pain points. The sessions are interactive and are meant to have continuity between the problem and a potential solution.

In the case study provided later in this paper, we decided to improve a process called “Level 3 Escalation Process”, where the customer support team escalates a problem to the development staff. For the customer role, in this case, the customers were the support agents that initiate an escalation.

4.2.2 Innovation Catalysts

Innovation Catalysts are people who are trained in D4D facilitation, and help the team with the process. An Innovation Catalyst may or may not be part of the team. They

typically plan the session with the team leader, make sure the room logistics are prepared, and conduct the session with the team.

The Innovation Catalyst will train the team about the methodology, keep time, and generally guide the team through the day. People trained in the art and science of innovation are very important to the success of the program (Tenant Snyder 2008).

4.2.3 The Team

The team is typically sized at 4-5 people, small enough to stay fully engaged while large enough to generate a diversity of ideas. The Innovation Catalysts like to quote the "2 pizza" rule, where the team has to be small enough to be fed with two large pizzas.

The team for a typically D4D product session might consist of Product Managers, Experience Designers, and Software Engineers. In our process improvement sessions, we included Project Managers, Software Engineers, and Software Quality Analysts.

4.3 Deep Customer Empathy

The goal for Deep Customer Empathy is to know the customer and their pain points better than they know themselves. This involves direct observation, interviews, process mapping, affinity diagrams, and other techniques to learn about the opportunities to help improve the customer's experiences.

As mentioned in the roles section, it's important to involve actual customers in these sessions. In a typical product-oriented D4D session, four to five customers would be included.

IDEO is one of the pre-eminent design firms and they have published a set of cards that summarizes many of their customer empathy techniques (IDEO 2003). These techniques include:

- A day in the life
- Affinity Diagrams
- Body Storming
- Camera Journal
- Card Sorting (and building mental models)
- Long Range Scenarios
- Role Playing
- Shadowing

The Stanford Design School has published a set of descriptions for their design techniques, with some overlap with the IDEO set. The Bootcamp Bootleg can be downloaded from their web site (http://dschool.stanford.edu/) (Stanford 2010).

The Pacific Northwest Software Quality Conference proceedings from the 2010 conference has a paper about Customer-Driven Quality, which describes many tools and techniques for incorporating customer insights in a software development life-cycle. Many of these techniques are applicable to gain customer insights for improvement activities. Among these are “Follow Me Home” and “Voice of the Customer” (Ruberto 2010).

4.4 Go Broad to Go Narrow

The “Go Broad to Go Narrow” stage is where creativity happens. One tenet is that to come up with a great idea, you first need many ideas to choose from. The Innovation Catalysts like to see 50 to 80 ideas generated.

Brainstorming and idea generation is encouraged to be as crazy and creative as possible. Often a crazy idea will spur a new line of thought. For example, an idea of giving away the product for free may sound crazy to business people, but may lead to new ideas in business models, such as a free trial.

Once there are a sufficient number of ideas, they are combined and evaluated for a few to be tested through Rapid Experimentation with Customers. The Innovation Catalysts help select the criteria to evaluate the solution ideas, with speed of implementation typically being part of the criteria.

4.5 Rapid Experimentation with Customers

Having a few ideas in hand, the team tests those ideas with the customers. To make the test more accurate, the idea is fleshed out into a prototype or at least a storyboard. A storyboard can be drawn in a few minutes. It’s a few minutes by definition, because the Innovation Catalysts allocate only a few minutes to creating the storyboard. This gives the team time to complete several iterations with the customers.

5. Design for Delight Applied to Process Improvement

Our team applied the D4D process to improve our processes. This section illustrates how we did that with an example.

5.1 Choosing a Process to Improve

Any discussion of process improvement should start with a discussion of processes. Books have been written on process standards and frameworks (Persse 2006). For our purposes, we will consider a simple framework as shown in figure 1.

Figure 1: A simple process framework

We used the D4D process to improve the “Level 3 escalation process”. This is the process where the customer support team escalates problems to the Product Development organization, for analysis and potential correction. Our support team fields calls from our customers, and occasionally a single problem affects several customers. When this happens, the support team escalates the problem to the development team. In the past, this escalation process has been error prone, and communications between the support and development teams had a high noise ratio.

We used D4D to bring in support agents (the customers of this process), to understand their side of the process, their pain points and constraints. Then, followed the D4D process to generate a lot of ideas, evaluate those ideas, and narrow down to the few improvements that would make the most impact. Finally, we prototyped the improvements and tested those with the support agents before investing a lot of resources, so we could choose the most effective corrections.

5.2 D4D logistics

The Design for delight process works best with all of the participants together in a room, with tables, whiteboards, flip charts, plenty of post-it notes, and markers. We spent about 6 hours for the first session, which included an orientation on the overall process, a debrief afterwards, and lunch in the middle. Four hours should work well for a second session, where there may be less time required for training and debrief afterwards.

5.3 Process Mapping (Deep Customer Empathy)

The first step of D4D is “deep customer empathy”, which means that we should understand the customer’s emotions about the process that we are improving. But before we can understand the emotions involved, we need to gain a common understanding of the existing process steps. We did this in collaboration with the support team. The support manager used a whiteboard and post-it notes to lay out the process.

He wrote down each step on a Post-it(tm), and placed it on the board and used a marker to draw the process flow, branches, and loops. The QA team participated as well, asking clarifying questions, adding their own observations, and filling in details.

The emphasis is strongly on capturing the process as it exists in practice, not as we think it should or wish it to be.

5.4 Pain point identification (Deep Customer Empathy)

Once the process was laid out, we asked the support manager to evaluate the process steps in terms of pain and pleasure. Assigning an emotion to each step of the process involves both sides of the brain, left (logical) and right (emotional). Delight is right in the name of “Design for Delight”, and it’s an emotion. We want our solutions to be built on addressing the emotional aspects of process improvement.

Steps that caused pain for the support team were moved to the bottom of the board, while steps that appeared to be working well were moved toward the top.

We made sure that everyone understood the pain points, and then chose one of the pain points to address in the problem-solving phase.

The QA team already participates in the level 3 escalation process, so they were somewhat familiar with the pain points raised by the support manager, but this discussion allowed them to see the pain in a new light, from the support manager’s perspective.

5.5 Brainstorming (Go Broad to Go Narrow)

Brainstorming solutions is where the delight in D4D is born.

Remembering the “two pizza” rule, a good team size for this stage is four to five people. We spent about 15 minutes brainstorming ideas, suggesting people to come up with the most outlandish solutions possible. The extreme ideas are not usually fruitful in themselves, but often lead to new thinking when shared with the team.

5.6 Evaluating, grouping, selecting (Go Broad to Go Narrow)

Before we started to “go narrow”, we had to go a little broader. Each person took their ideas and one by one posted them to the board. As they posted them, they read each idea out loud. The other participants actively listened, asking clarifying questions, and made connections to their ideas, or even created new ideas on the fly. They are not judging the other persons’ ideas, but to building upon them.

As people posted their ideas, they were grouped with similar ideas as themes came out. Once everyone had posted his or her ideas, then the group took another round of brainstorming.

During this phase, the innovation catalyst asks several questions to spur thought:

- What about this idea would make of the experience great?
- How can I improve on this idea?
- What would we never even think about doing?

This stage takes another 20 minutes, combining ideas and feeding off each-other's ideas. After that, it's time to start to evaluate and narrow down from many ideas to a few that will be tested.

We used a 2x2 grid to start to narrow down the ideas. We drew the grid, depicted in figure 2, and placed each post-it note with an idea into the grid. This evaluates each idea in two dimensions, speed of implementation and impact to the solution.

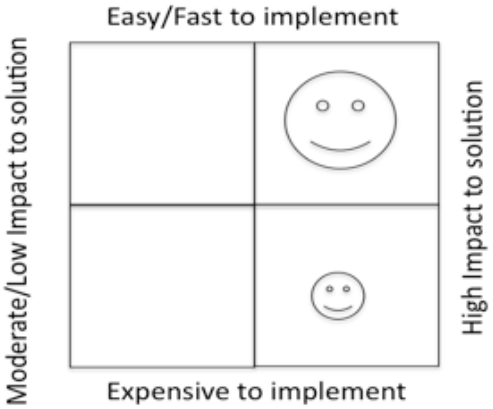


Figure 2: Grid to assist in idea selection.

This evaluation method helps to identify ideas that are both likely to solve the problem, and allow for rapid experimentation (ease or speed of implementation).

If there are many ideas in the upper right quadrant (fast to implement, with high impact for the solution), the innovation catalyst leads the team in a prioritization exercise, forcing the team to list each idea in priority order. This is frequently done by expanding the quadrant and re-judging each idea relative to each other. The top 2 – 3 ideas are fed into the next stage of D4D, rapid experimentation with customers.

5.7 Storyboarding (Rapid Experimentation with Customers)

Storyboarding is another technique from the design world. Storyboarding depicts the newly improved process as a sequence of drawings that tell a story, like a comic strip. Telling the story of how the new process will work will put the customer in a mindset of visualizing how the new process works, and allowing him/her to better judge the effectiveness of the solution. This provides the opportunity for more realistic feedback than a verbal description alone.

Figure 3 shows an example story board, which coupled with narration describes the new process for the customer to evaluate.

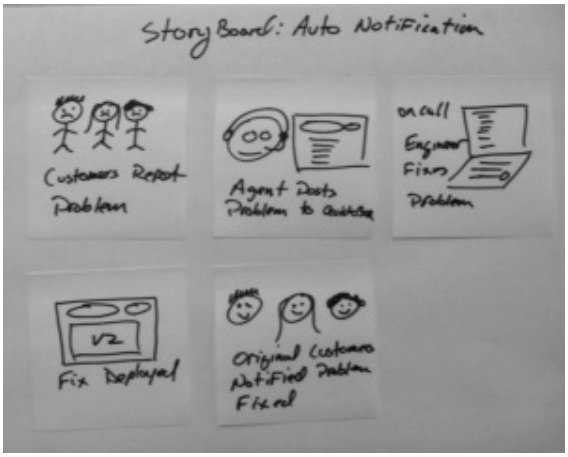


Figure 3 - An example story board

After generating a lot of ideas for improving the Level 3 escalation process, and selecting a few to prototype, we created storyboards to describe the change. We brought the customers back into the session and presented each storyboard – taking care to describe the scenario.

The customers asked a few clarifying questions and made comments. One story was completely off mark, while another was pretty close. We were able to get customer feedback and refine the solution with about 40 minutes of time spent.

5.8 How Did D4D Work for Process Improvement

The teams participating generally supported using this process for process improvement. A survey conducted a few days afterwards resulted in 8 of 9 participants indicating they would recommend D4D as a choice for process improvement. Some of the reasons cited for this recommendation were:

- A comprehensive set of ideas were generated, resulting in creative solutions
- The nature of the process was very collaborative
- Several “ah-ha” moments in realizing the customer’s pain, from his/her perspective.

Some of the areas for improvement expressed in using D4D:

- The results of using D4D are highly dependent on which customer(s) you include. These may or may not be the most important pain points to solve, but may reflect what was foremost in their mind that day.

- The results tended towards minor and incremental improvements – not the dramatic “reimagining” of the underlying process. This is largely determined by the selection criteria used, especially the factor where we evaluated those ideas that were quick to implement. Care should be taken to choose the selection criteria, or after the rapid experimentation phase to follow up and evaluate those ideas that would take longer to test.
- Evaluation and selection of ideas to pursue tended to be based on group consensus. There is great power in having relatively uninformed people participate in the brainstorming sessions to generate fresh thinking, but it didn’t seem as accretive in the selection process.

6 Summary

D4D provided a framework to engage with process owners and generate some creative solutions. It certainly engaged the team with process owners in a fun and energetic session. The results are highly dependent on the customers that are engaged, the pain points they list, and the selection criteria used to evaluate ideas.

References

ABC Nightline (TV program), Smith, Jack (correspondent); Deep Dive, July 1999

IDEO, IDEO Method Cards, 51 Ways to Inspire Design, William Stout Architectural Books, 2003

Kelley, Tom, The Art of Innovation, Profile Books, 2002

Martin, Roger L, The Innovation Catalysts, Harvard Business Review, June 2011

Persse, James R., Process Improvement Essentials, O’Reilly Media, Inc., 2006

Ruberto, John, Using Customer Driven Quality to Manage Complexity, Pacific Northwest Software Quality Conference, 2010

Schwaber, Ken, Agile Project Management with Scrum, Microsoft Press, 2004

Spear, Steven J., The High-Velocity Edge, McGraw Hill, 2009
Spear, Steven J., Operational Excellence: From Fragmented Vocation to Principle-Driven Profession, 2011

Stanford Design School, Bootcamp Bootleg, v2, 2010

Tennant Snyder, Nancy; Duarte, Deborah L.; Unleashing Innovation, How Whirlpool Transformed an Industry, Jossey-Bass, 2008

Vogel, Craig M; Cagan, Jonathan; Boatwright, Peter; The Design of Things to Come: How Ordinary People Create Extraordinary Products, Pearson Prentice Hall, 2005

Releasing Software
(How do you know when you are done?)

Doug Whitney
douglas_whitney@mcafee.com

Abstract

Releasing software. How do you know when you are done? There are several items that can be added to a release checklist, but that last one may take so long that it delays shipment. Did you do the legal check? Have you completed an internal deployment? The release criteria apply to project management, program management, development, quality assurance, publications, localizations and support. It happens throughout the entire lifecycle and should not wait until the project moves into the final testing phases. There are also methodologies that go beyond the release itself, like, internal deployments, phasing out the release to a small set of customers, and product supportability. There are many more questions and we will never get it perfect, but the aim of this paper is to help you to find out what you can do to enhance the release process for you.

Biography

Doug Whitney is a development manager for one engineering team and program manager for four other teams at McAfee. He has presented papers at PNSQC and Quality Week on various topics. He has 19 years of engineering management experience and has managed QA teams at both McAfee and Intel.

Introduction

My son is a 9th grade student and had an art project that was a pencil drawing enlargement of a photograph. It was a man playing the guitar. He showed it to us and, being the proud father, I said it was wonderful. He responded, “But dad, I am not done”. He was not ready to “release” it to his teacher for a grade, even though he “tested it” on his mother and I. So I asked into this and he indicated he needed to erase some of the area on the guitar and change the background. The idea of testing a release on a small sample is nothing new. To have checkpoints along the process to validate that all necessary requirements and mandates are completed, is a necessary part of what we do. This is not just in a school art project, but also in releasing software. Here is his final picture, not bad for 14.

This paper is about the process we use at McAfee to release software. It will touch on the checklists we use when releasing, the entrance and exit criteria we use for milestones, and the idea of phasing out the software both internally and externally.

The checklists

At each milestone, there is a checklist that validates what needs to be accomplished prior to moving on to the next one. We also have a milestone checklist for the planning processes we use at McAfee. Our product lifecycle framework (PLF) spreadsheet has different tabs for each of the milestones.
Items to consider at each of the milestones ***that can influence*** the release:

Plan commit milestone checklist:

- Identify any legal requirements for 3rd party software being used
- Milestone release criteria established
- Identify software and hardware requirements and timelines for acquiring
- Localization plan reviewed and scheduled
- Risks identified with mitigation steps
- Testing and defect status reporting established
- Rollout schedule created (Security as a Service - SaaS)

Code complete milestone checklist:

- Sign off from legal for requirements
- All required software and hardware is available
- Localized drop completed
- Internal deployment requirements reviewed and approved
- Release to Support plan initiated (corporate)
- Testing and defect status report presented weekly
- Security Audit complete

Release candidate milestone checklist:

- Bill of materials created
- Code analysis completed
- All required external approvals received (Windows Hardware Quality Lab (WHQL), Win Logo certification, etc)
- Product compatibility testing completed
- Live Virus/GOAT (sacrificial goat – think Jurassic Park) testing completed
- Readme file sent to pubs for final review and localization
- Release to Support plan completed and reviewed
- Supportability document draft created
- Performance / Soak tests completed and results meet expected gains

McAfee has made a strong effort to create a comprehensive checklist that will reflect what needs to be completed at each milestone. I have attached the actual checklist items in the appendix. The checklists are broken down by each functional group and have sections for product management, program management, development, QA, technical publications, support and localization. Not all groups have activities at each milestone, but some have activities in all of them.

Testing Activities by PLF Phase & Milestones

Iteration Testing

Goals

The goal of this testing is to validate that the functionality of the feature delivered as part of the iteration. It validates that the base level of functionality defined for this application is stable and that new functionality added for this release has not destabilized the product.

This test pass will:

1. Verify that the product's pre-defined Build Verification Test (BVT) passes with the "Iteration complete" build.
2. Verify that the product is stable enough to support more detailed testing.

Entry Criteria

1. The development manager declares that a "Iteration complete" build for the iteration is available. In this build, all new or modified features and functionality specified for this release are fully implemented and ready for testing.
2. Any bug fixes or enhancement requests targeted for this release, and specified within the release planning documents are fixed and included in the 'feature complete" build.
3. The "iteration complete" build is available to QA as an official build.
4. The BVT for this product is updated based on delivered functionality
5. The Bugzilla Defect database has been setup for this product.

Exit Criteria

1. All tests within the product's BVT pass
2. All test plans required for the iteration testing phase of the project are complete, reviewed, and approved.
3. All detected defects have been logged in Bugzilla.

Beta Testing

Goals

The goals of this test pass are to demonstrate an increased level of stability as defects found in the previous test passes are fixed, and to increase the level of compatibility testing by including secondary languages and platforms in the test cycle.

This test pass includes:

1. A complete execution of functional test plans associated with each area of the product.
2. A complete regression test of all current test plans
3. A complete pass through all defined integration tests.
4. Compatibility testing of primary supported languages.
5. Conduct ad-hoc testing will be performed on application
6. Focus on resolving beta feedback issues
7. Completion of Live Virus Testing.

Entry Criteria

1. The schedule and test plans have been reviewed to determine if, based on the results of the previous test pass, the testing planned for this pass should be increased or reduced.
2. The project team has agreed upon any required changes to the testing for this pass and the schedules have been adjusted accordingly.
3. Development has executed a Build Acceptance Test (BAT) and has unit tested all bug fixes against the Release.
4. Development of all required test plans and procedures for this test pass is complete.

Exit Criteria

1. All planned tests have been run.
2. All detected defects have been logged in Bugzilla and scrubbed
3. No test blocking bugs exist which would prevent entry into the next test pass
4. All “Beta Critical” defects have been resolved.
5. All fixed defects have been included in an official build and verified.
6. All Severity 1 defects fixed in the release to date have been re-verified, and found to be fixed, in the Beta candidate build.
7. Pseudo-translation builds to be tested (Internationalization -I18n)

RC (Released Candidate) Testing

Goals

The goals of this test pass are to complete the planned testing for this release. At the end of this test pass every planned test will have been run on at least one Operating System (O/S) or platform combination.

This test pass includes:

1. Focus on regressing previously discover defects and issues
2. Conduct ad-hoc testing will be performed on application
3. Successful completion of Live Virus Testing.
4. Compatibility testing of primary supported languages.
5. Continued testing of various O/S and languages.

Entry Criteria

1. All beta feedback has been incorporated into the product or otherwise responded to.
2. All Severity1 and 2 issues have been resolved.
3. Development has executed a Smoke Test and has unit tested all bug fixes against the Release.

Exit Criteria

1. All planned test plan have been executed.
2. The product has been tested all planned platform/browser/language combinations.
3. All detected defects have been logged in Bugzilla and scrubbed.
4. All “Release Critical” defects have been fixed, verified, and closed.
5. All fixed defects targeted for this release have been regressed.
6. Fixes for all severity 1, and priority 1 defects fixed in this release have been re-verified with the RC build.
7. Incoming defect rates have been declining for at least 2 weeks prior to this milestone.
8. Key beta testers agree that the product is ready to ship.
9. Product has successfully passed Live Virus Testing recognized by McAfee Labs

RTW (Release to World) Testing

Goals

The goals of this test pass are a final validation of the product. This is primarily to verify any fixes included in the RC candidate build, and to ensure that those fixes have not destabilized the product.

This test pass includes:
1. Final validations of the RC candidate build.
2. Verifying and closing any defect fixes made in the RC candidate build.

Entry Criteria

1. An official RC candidate build has been created that includes all defect fixes targeted for this release (i.e. no unfixed, release critical, defects remain).
2. Development has executed an IBT and has unit tested all bug fixes against the Release.
3. Release documentation has been created, reviewed and approved by QA

Exit Criteria

1. All code changes made since the RC milestone have been inspected by a cross-functional team.
2. All defects fixed in this build have been verified and closed.
3. All testing targeted for this release is complete with summary reports available.
4. Product release notes have been inspected.
5. Agreement from all functional groups that the product is ready to be released.
6. Release Plan, Process, and Schedule created and ready for use
7. Zero article testing (download and install as a customer would do) is complete and passed to the Deployment and Integration (DI) Team)

RTM (Release to Manufacturing) Testing

Goals

The goals for the RTM phase is to complete the validation of the final RTW build as it is integrated into the retail CDs.

This test pass includes:
1. Validation of the CD images and compliance to RTM Bill of Materials (BOM).
2. Release of Master CD's to manufacturing.
3. First Article Inspection completed.

Entry Criteria
1. Complete and signed off RTM BOM.
2. No product fixes or changes made to the build included in image.
3. ISO images pass initial Acceptance test.

Exit Criteria

1. CD images meet the requirements established in the RTM BOM.
2. 100% of planned testing competed.

No trade mark violations and correct documentation and PP included on CDs.

Phasing out the release

McAfee in Action

The internal deployment of enterprise products within McAfee is referred to as “McAfee in Action” (MIA). This is not to be mixed up with McAfee inaction. ☺ McAfee releases its software internally through a series of deployments that are geographically based. We have offices that conduct these deployments in Beaverton, Santa Clara and Plano Texas. There is a MIA coordinator who manages the expectation of the IT groups, who actually do the deployments, as well as manages the schedule for rollouts. This type of deployment allows for the build to be consumed by a professional IT organization prior to deploying to the world.

The phased rollout

The phased rollout approach is where we upgrade a subset of customers and wait to see if there are issues. We use this approach in several ways depending on the type of software.

Enterprise
For our enterprise customers, we have a process in our lifecycle called RTS, or release to support. The support engineer then will work with our development engineers to release to a subset of customers we refer to as the early adopters. We have goals for Early Adopter Program (EAP) that are product specific. Some products will want to roll to 100 customers and others will want to roll to 100,000 total nodes. Engineers will often go on-site with these customers to ensure that any rollout issues are understood and documented. Once the issues are documented and any critical ones addressed, we will make the software generally available by moving it to the download site for remaining customers.

SaaS model
McAfee has had SaaS model software offerings for over 9 years. The phased rollout approach we use is based on configurable settings on the NOC (Network Operating Center) that can be modified to upgrade based on the customer identifier in our database. We will start with all new installs and by identifying customers that will provide us with a total of 5000 nodes. We usually find ones that do not have custom policy, or may be on an OS platform we have supported for some time. Once we have the customers identified and they rollout, we stop and determine if there are any calls to support. We also use a method where the installer will report success or failure during the install process and upload a small response to the NOC. We can then query this data for success or failure. The install status reporting is a switch that can be configured since we do not need this operation running during normal operation, only during the phased upgrade process.
The phases are established and documented in a phase rollout plan that is part of the initial planning documents we use to get to a plan commit status in our product lifecycle. We start with all new customers. We then roll out to 5,000 upgrades, and then a week later to 15K more. We then take a break for about a week and assess issues. If there are no issues that need to be addressed, we will move forward with 60K, then 125K, then 250K and then open it up for the remaining customers. We also plan an update to the code and schedule it as a potential production refresh.

Consumer
The consumer team approaches things differently. They start with Australia. The release of software starting with Australia actually provides many advantages.

- They are in a time zone that is close to one of our larger development center in Bangalore.
- The development center and call center in Bangalore are in close proximity
- It is a statistically valid sample
- They speak English in Australia

At the call center, we will have engineers on site during the rollout. If there is an issue raised on the phone that is not known, the engineer will listen in on the call and determine if there is additional action or information needed. In all cases we manage the information received and stop the rollout if critical issues are discovered. Once fixed, we begin the cycle again.

Conclusion

When the areas that are time sensitive are scheduled early and acted on, it reduces the risk of a late deliverable. When each milestone has clearly defined entrance and release criteria, it will reduce the risk of items being carried over to the next milestone (or released). When the internal deployment or a phase release discovers an issue, it reduces the risk of it being seen broadly in the field. These are approaches that can be used to help answer the question, “How do you know when you are done”.

Appendix

Example of SaaS rollout in table format:

Partner	Languages	RTQA	RTW (Go Live Date Client & NOC)	Comments
Pre-Phase 1				
McAfee - Beta	ENU		5/21/2009	ENU Only
Beta Refresh	ENU	7/14/2009	7/16/2009	RTW bits deployed to Beta
Rollout Phase 1				
McAfee - Phase I-A	All	-	7/7/2009	All "New Installs"
McAfee - Phase I-B	All	-	7/14/2009	5K English only - no custom policy - No Browser Protection
McAfee - Phase I-C	All		7/16/2009	15K English only - no custom policy - No Browser Protection
Potential Production Refresh	All			Production refresh deployed
McAfee - Phase I-D	All		8/20/2009	Restart of the Rollout 20K English only - no custom policy - OK Browser Protection
McAfee - Phase I-E	All		8/27/2009	60K
	All		9/22/2009	125K
McAfee - Phase II	All	-	9/24/2009	260K
McAfee - Phase III	All	-	9/29/2009	Remaining users

Concept Commit Checklist

Function	Item or Deliverable
Product Mgmt	
	Present any critical Feature Modification Requests (FMR), new platform support / service pack or 3rd party compatibility testing
Program Mgmt	
	Patch Milestones created to be tracked in Health Checks and Plan of Record (POR) meetings
Development	
	Consumed component team has been made aware of the timelines and there are no issues
	Engineering Statement of Work (SOW) - DEV estimate complete on high, medium and low defects
	Initial SOW complete (identify priority 1,2 and 3 defects for inclusion)
QA	
	Engineering SOW - QA estimate on backend schedule with a quarter level timeline for Release to Support (RTS)
	Initial SOW complete (identify priority 1,2 and 3 defects for inclusion)
Tech. Pub's	
	Commit from Tech Pubs for Readme Review / Sign off
Localization	
	Engineering SOW reviewed by Localization
Support	
	Agreement on prioritized SOW
	Top customer issues have been identified

Plan Commit Checklist:

Function	Item or Deliverable
Product Mgmt	
	Schedule reviewed & approved
Program Mgmt	
	Identify any legal requirements (3rd party exports, etc.)
	Master Schedule reviewed & approved
	Milestone Review Plan completed for the project. This plan consists of a copy of this spreadsheet with the milestone criteria tailored to suit the specific project
	Milestone Review Plan completed for the project. This plan consists of a copy of this spreadsheet with the milestone criteria tailored to suit the specific project
	Project entered into Bullseye
	Project Launch Meeting with full team
	Risk Analysis Matrix
	Risk List Generated for weekly tracking
Development	
	Build System Setup to start development and coding (e.g. SVN repository, Branching Plan, Team City etc.)
	Development Schedule reviewed & approved & communicated
	Development Plan reviewed & approved
	Functional Spec's or equivalent reviewed & approved (If using Sustaining Agile specify the Iteration Plan if applicable, i.e. number, duration, etc)
	Integration Plans Created and signed off between product teams (common, shared components: VSCore, McAfee Foundation Services (MFS), HIPSCore, etc.)
	Merge plan draft has been completed to identify any mainline forward merges
	Post Concept Commit requirements qualified
	Required hardware identified and ordered
	Required software identified and ordered
	Required training identified and scheduled
QA	
	Automation Plan has been created, approved and resourced
	Bugzilla product & version created & accessible
	Content Security Testing (Host Intrusion Prevention (HIP) Content, Live Virus Testing (LVT) and GOAT testing approved)
	Project entered into Bullseye with key milestones setup, risks, dependencies
	QA Plan reviewed & approved & communicated

	Required hardware identified and ordered
	Required software identified and ordered
	Required training identified and scheduled
	Send out notification all parties on schedule and plan
	Test Case DB setup and Functional Decomposition updated
Localization	
	Localization Plan reviewed & approved
	Localization Schedule reviewed & approved
Support	
	Schedule reviewed and approved
	SOW plan has been updated on tier 3 central

Feature Complete Checklist:

Function	Item or Deliverable
Program Mgmt	
	Sign off from legal on any 3rd party inclusion
Development	
	All P1's Customer Fixes implemented and any deltas communicated to support and Product Management
	Drop to Localization team
	Feature complete build (BVT Passed) delivered with Software BOM & release notes
	Identify defects for merging
	Software BOM created
	Unit tests complete and passing, including report on coverage
QA	
	All reported defects have been scrubbed in Bugzilla
	All required HW & SW is available
	All required test scripts reviewed, and approved
	BOM validated by QA
	Bullseye status is being updated at least weekly
	BVT Passing 100%
	Determine what content should be in the readme
	Install / Deployments tests complete
	McAfee In Action Initiation Document reviewed and approved
	Scalability lab testing complete, results verified as expected
	Testing and Defect Status Report presented weekly (e.g. Tiki, Twiki)
Support	
	RTS draft plan is created

Release Candidate Checklist:

Function	Item or Deliverable
Development	
	All static analysis reported bugs have been reviewed & responded to
	All deferred defects have been reviewed by the cross-functional team (Sustaining DEV & QA, Support, PM)
	All fixes that are common with current mainline code base have been merged per Standard Merge Process
	All open defects have been fixed or deferred
	Complete list of included common McAfee components (e.g. Engine, Agent, Virus Scan Core) has been created. Included versions have been confirmed as releasable by component team
	Patch Supportability Document Draft created
	RC build delivered with Software BOM updated & release notes
	Source code escrowed
	Unit tests complete and passing, including report on coverage
QA	
	All Bugs Verified and Smoke / Stability tests performed around components affected; Bugs verified on at least 2 Platforms
	All planned testing complete with results recorded and 100% pass rate. Any failures have associated defects logged in Bugzilla, with the defects marked as "deferred"
	Any required external approvals have been received (e.g. Windows Logo Certification)
	Bullseye status is being updated at least weekly
	Compatibility testing with other McAfee products complete. This includes testing will all products sold in a suite with this product. (full Point Product Compatibility test (PPCT) testing, i.e. ePO, McAfee Agent)
	Exploratory and Area testing completed around affected components
	Final package has passed an complete in-depth Extended Build Validation Test (eBVT)
	Final package marked in eCM with milestone
	Final package validated against the BOM
	Goat testing passed with results approved by Avert (AV Scanner products only)
	Install/Uninstall testing confirms a complete uninstall (verified with "beyond compare" or similar tool)
	LVT passed with results approved by Avert (AV Scanner products only)
	Patch tested in EAC setup
	Performance / Soak tests completed and results meet expected gains (equal to or better than previous patch depending on code changes)
	RC Build sent to MVT team
	Readme draft sent to tech pub for final review

Tech. Pub's	
	Readme's are localized
Support	
	RTS Phased plan completed and reviewed by team

Release to Support Checklist:

Function	Item or Deliverable
Program Mgmt	
	All sustaining PLF milestones reviewed and complete
Development	
	All static analysis reported bugs have been reviewed & responded to
	All fixes that are common with current mainline code base have been merged per Standard Merge Process
	All Priority 1 and reopen defects are resolved
	Any defects fixed since RC have received a detailed cross-functional code walkthrough and review of appropriate testing (DEV, QA, Support)
	Any MIA ship stopping defects have been fixed and unit tested
	Appropriate deferred defects identified and move to next patch or mainline release
	RTS build delivered with updated Software BOM & release notes
	Supportability initial draft reviewed
	T3 Support training completed as needed
	Unit tests complete and passing, including report on coverage
QA	
	All escalation fixes merged into mainline have a defect created for current mainline release
	All fixed defects are verified & closed
	All RTS builds have been promoted to RTS status within the build system
	All site deployments planned and at least 80% successful install with no showstopper issues
	All testing defined in test plan is complete with any failures approved
	Any open/deferred defects are scrubbed and disposition complete
	Bullseye status is being updated at least weekly
	Final package has passed an complete in-depth eBVT
	Final package marked in eCM with milestone
	Final package validated against the QA BOM
	Final readme reviewed and packaging complete
	Final report is updated in Twiki or central location
	Install/Uninstall testing confirms a complete uninstall (verified with "beyond compare" or similar tool)
	LVT passed with results approved and signed off by McAfee Labs
	MIA Closed, with all reported issues fixed and verified or deferred
	MIA Sign Off (IT, MIA Admin)
	Non-English builds are fully tested and ready for RTS

	Performance figures validated on release build without regressions (If % improvement expected then this should also have been met)
	RTS build has been tested against McAfee Virtual Technician (MVT) and passes
Tech. Pub's	
	Patch readme has been reviewed and signed off by tech pubs
Localization	
	Documentation complete & localized as required (including readme & release notes)
Support	
	Any updates made to RTS plan based on MIA or final testing
	Addendum Release notes Knowledge Base (KB) is ready

Release to the World Checklist:

Function	Item or Deliverable
Program Mgmt	
	All PLF milestones updated
	Retrospective scheduled for the project
Development	
	Any ship stopping defects found during RTS have been fixed and unit tested
	RTW build delivered with Software BOM & release notes
	Supportability document completed, reviewed, & approved
QA	
	All release packages have been posted to back up server for AV testing and storage
	All RTW builds have been promoted to RTW status within the build system
	Any multi-langauge testing which was not completed at RTS should be finalized
	Any ship stopping defects found during RTS have been verified
	Internal tracking tool (Bulls Eye) updated with final status for patch project
	Final package has been posted to the download site & download testing is complete
	Final package marked in eCM with milestone
	Final package validated against the BOM
Support	
	Readme KB addendum has been updated with any issues found in RTS
	RTS Phased release completed and no ship stopping issues found (at minimum 80% deployment with 100% success rate)

Inspiring, Enabling and Driving Quality Improvement

Jim Sartain
Jim_Sartain@McAfee.com

Abstract

This paper discusses some approaches used at McAfee, Adobe Systems and Intuit to drive continuous significant quality and engineering process improvement through the adoption of engineering best practices including Team Software Process (TSP), Scrum, Peer Reviews, Unit Testing and Static Code Analysis. It covers what has worked well and some of the challenges encountered. Key success factors and an overall methodology are outlined including:

- A multi-year improvement approach that can work in large organizations that may include both eager adopters and resisters of software quality and engineering process improvement.
- Customer satisfaction and defect removal metrics, including what is world class vs. what is typical, and the best practices that can drive major quality and engineering process improvement

Biography

Jim Sartain is a Senior Vice President responsible for World-wide Product Quality at McAfee, the world's largest dedicated security software company. He leads a team responsible for inspiring, driving and enabling continuous quality improvement across McAfee world-wide. Prior to joining McAfee, Jim held quality and engineering process improvement positions at both Adobe Systems and Intuit where he helped drive significant quality improvements in products such as Acrobat, Photoshop and QuickBooks. Jim also worked at Hewlett-Packard for seventeen years in a variety of software engineering and product development roles. His last job at HP was in the role of CTO/CIO for an Airline Reservation Systems business that serviced low-cost airlines including JetBlue and RyanAir. Jim received a bachelor's degree in computer science and psychology from the University of Oregon and a M.S. degree in management of technology from Walden University.

1 The Business Opportunity for Quality Improvement

Many software development projects are destined to be cancelled, significantly delayed or end up being delivered with a level of functionality, reliability or usability that reduces customer satisfaction. Even projects that deliver software may spend a significant portion of development effort on defect driven rework. Some projects spend more effort in finding and fixing defects than feature implementation. In fact, it is not uncommon in the industry to see teams consume more than half their overall development effort on testing and defect repair. Development organizations that make effective use of early defect removal methods including peer reviews and static analysis can reduce their rework costs to less than 10 percent. These world-class teams invest ~10% of their project effort in the early defect removal methods and deliver software on a frequent basis (e.g. monthly) that is at or near release quality.

Another significant driver of software quality costs is customer service and support expenses. Often, a majority of customer service costs are driven by usability, reliability or performance issues. The customer experience can be improved and support costs significantly reduced by identifying and eliminating the root causes for the most common product issues. Post-release issues are another common driver of rework that may consume up to 50% of organizational capacity to deliver post-release fixes.

The direct costs of poor quality (e.g. engineering rework, customer care costs) are usually the tip of the iceberg for quality improvement opportunities. Westinghouse Electric Corporation found that their indirect costs of quality (e.g. reduced sales, less time for innovative work) were three to four times the directly measured costs of poor quality (Campanella 1999).

Figure 1: Quality Improvement Opportunities

2 Net Promoter Methodology

Software development teams should have a way of understanding what is most important to customers and how they are doing with delivering great customer experiences. The Net Promoter Score (NPS) methodology is an excellent approach for doing this. The NPS methodology was developed by Satmetrix, Bain & Company and Fred Reichheld and is described in Reichheld's book the *Ultimate Question* (Reichheld 2006). There is more information at http://www.netpromoter.com.

When using NPS customers are asked about their likelihood to recommend a product or service. Delighted customers, called “promoters”, are so satisfied with their overall product experience and relationship with their supplier that they rate their likelihood to recommend as nine or 10 on a zero to 10 scale. Promoters will tell their friends, families and business associates to buy and use the product and they are responsible for generating up to 90% of positive product referrals. Customers rating their likelihood to recommend from zero to six are considered “detractors” and are responsible for up to 90% of negative word-of-mouth. Social media (e.g. twitter, blogs) provides an increasingly visible means for product promoters and detractors to share their views with thousands of customers world-wide.

The NPS methodology includes an all-important follow-on question: “What change would make you more likely to recommend the product?” Product teams use these questions to prioritize development priorities and to continuously improve the customer experience. To calculate NPS you take the percentage of customers that are promoters and subtract the percentage of customers that are detractors. A world-class NPS for commercial software products is 60%. An example of an NPS score of 60% would be having 70% promoters and only 10% detractors. Figure 2 illustrates the NPS calculation.

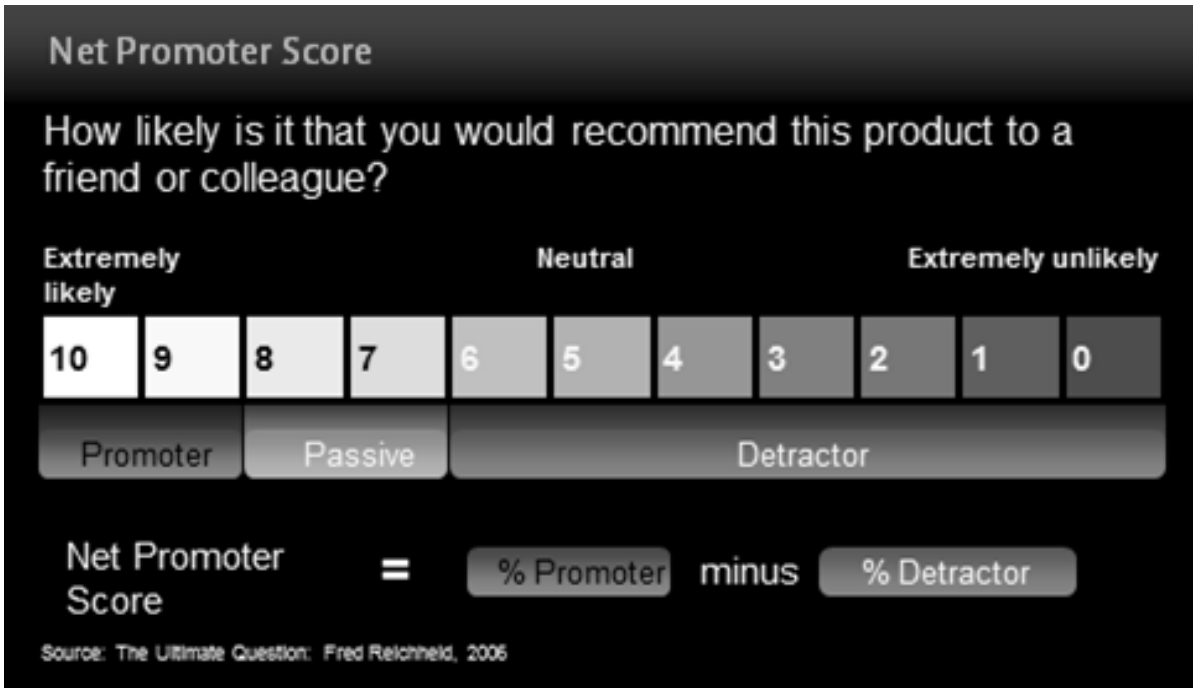


Figure 2: Net Promoter Score Formula

3 Quality Improvement Vision

To drive improvement there must be a vision describing how much better things can be when significant quality improvement are achieved. The vision is a tool to help ensure quality is a priority at all levels of the organization from senior leadership all the way down to front-line employees. The vision should include compelling benefits for key stakeholders of quality including customers, employees and shareholders.

Having an improvement vision that includes continuous improvement is important even for teams that are currently delivering very good quality; otherwise a natural tendency is to accept the current

level of customer experience and organizational effectiveness as “good enough.” I’ve found that teams with the best current quality in an organization frequently are not the first to get onboard with mew improvement initiatives. The teams with significant quality issues, especially if they work directly with customers are usually the easiest to get buy in from. Organizations that work directly with customers, are the most likely to understand and feel the customer pain from product issues. Figure 3 outlines the McAfee vision for continuous quality improvement.

Figure 3: McAfee Quality Improvement Vision

3.1 Customer Benefits

The vision should emphasize directly engaging with customers to understand what is important to them and how you are doing with product quality. Leveraging this information a team can continuously improve to maximize the number of product promoters. In a 2010 study of its customers, Adobe found that that it’s most satisfied customers:

- Were more likely to recommend Adobe products to others
- Invested a greater percentage of their budgets with Adobe
- Were less likely to consider alternative solutions

3.2 Employee Benefits

Employee benefits should include improved work-life balance. Also, employees will benefit from having more time available to do valuable and more innovative work. Providing a less stressful work environment that offers more time to do truly innovative work is a great way to increase employee engagement and ultimately retention. I’ve seen teams at several companies free up 20% more time from reductions in defect-driven rework.

3.3 Shareholder Benefits

Improving product quality can increase the number of product promoters and decrease the number of detractors, which will lead to increased sales, customer retention and the ability to charge premium prices.

Also, productivity improvements from early and efficient removal of defects can free up capacity for work that has a higher return-on-investment. Organizations adopting a quality first paradigm can avoid a vicious cycle where they must spend an increasing amount of their efforts on customer support, software maintenance and bug fixing. Because they are spending an increasing proportion of time on those activities they may not be able to direct adequate resources to innovation. This can ultimately lead to a lack of competitiveness, business decline or even bankruptcy (Humphrey 2001).

Quality improvements will also benefit business efficiency. Customer care budgets can be decreased by reducing the number of customer service calls driven by software defects, usability issues or overly complicated business processes. Other business functions (e.g. legal, PR, engineering) are frequently impacted with the emergence of and aftermath from quality issues that reduce the reliability, usability or performance of products.

4 Required Leadership Support

Leaders are responsible for ensuring the highest priority for their organization is delivering high quality software that meets business goals which will typically include the delivery of quality software by a specified date. This is not a contradiction -- focusing on delivering quality software is the best way to ensure delivery dates are met. A frequent root cause for missing schedule commitments is finding an unexpectedly high number of defects late in the development cycle.

Most software engineers have a strong desire to deliver software of high quality and will if they are provided the right environment: one that includes strong support for quality from all levels of management. To ensure strong leadership support for quality, managers should have quality and engineering process improvement as part of their performance goals. To ensure that all teams aggressively drive to improve quality senior leadership should require all product development leaders to show how every release is measurably better than the last.

I remember a situation from my first Quality Leadership role. The Product Development leaders were asking me why the level of quality was not improving for their products. The simple truth was that it was not an organizational priority for them, and therefore their teams. They expected that the manager responsible for QA or testing and engineering process improvement would somehow be able to drive improvement without any explicit support from them. This was clearly a problem. The solution was that I asked our common boss to ensure that he held each Product Development leader accountable for showing measurable improvement. His initial response was that he was fine with this, but he didn't know how to measure their progress. My response was: "This is not your problem. Any leader that can't assess how their projects are doing with respect to measurable improvement is not doing an adequate job." Soon they were asking for my assistance in helping them identify ways they could measurably assess how they were doing with quality and quality improvement. These leaders now considered it to be *their* priority to improve quality.

4.1 Quality Improvement Objectives, Goals and Metrics

A quality plan including goals and a strategy for achieving them should be established at the start of a project and teams should have compelling quality improvement goals that include specific measurable goals and realistic improvement targets. Teams should also set their own improvement targets as this approach helps ensure ownership and commitment for achieving them. If targets are set tops-down by managers then there is a risk of the teams not buying into them because they don't "own" them. Or in some cases, teams will believe their tops-down targets are impossible to achieve. Any lack of confidence or commitment may not be discovered until it is too late to prevent a schedule

slippage and/or release quality issues. The quality goals must be clearly aligned with what is important to the organization.

To help set aggressive but achievable goals, it is useful to have benchmark metrics such as the percentage of defects found and removed using peer reviews. The best benchmarks are from other teams within the organization. The next best source of benchmarks would be from other organizations in the same enterprise. If this isn't possible, then industry data can be used (Jones 2010). Benchmarks closer to the team are not only more likely to be relevant, they are also more likely to be accepted by the team as valid. The collection and sharing of metrics across an organization will help provide relevant benchmarks. These benchmarks can help support a cycle of teams learning about best-in-class performance in the org and then raising the bar when they set their next set of improvement goals.

4.2 Quality Review Meetings

The senior leadership team must regularly review the status of software quality as a team. During these review meetings each team leader summarizes improvement progress and status relative to their quality goals and plans. This provides an opportunity to recognize teams making progress. By publicly recognizing progress, leaders reinforce the importance of improvement. In cases where progress is not being made there should be discussion as to why, as well as the commitment of necessary support to get improvement on track. Senior leader's regularly spending time discussing quality goals and improvement progress will highlight their importance. Also, these meetings serve as an educational opportunity for business leaders to better understand software quality economics. This deeper understanding enables the business leaders to ask the right questions and to be more specific (and genuine) in their praise about improvement progress.

Another benefit of these quality review meetings is that the teams that are make significant progress become role models for the rest of the organization (especially when they are publicly recognized for their progress). Many times I've seen first surprise and then support for improvement goals from other teams when they saw how well the best teams performed. For example, a common goal is to challenge teams to set a goal for the percentage of defects that are found and removed using early defect removal techniques such as peer reviews. When teams see other teams, working under the same constraints, pressures and same code base, finding 50%, 60% or even 80% of their defects early using engineering best practices then they are likely to set their own goals.

5 Engineering Best Practice Rollout Strategy

Most engineering best practices have been around for years and are known to many engineers as best known practices. So why aren't they common practice? One of the major reasons teams hesitate to adopt best practices is that changing how they work involves taking a risk. The conundrum is that teams often don't see the benefit of the "new way" and therefore become convinced unless they try it. The most credible recommender of a new practice is a team that already adopted it and seen practical benefits.

Also, the adoption of new best practices may require support from the entire team. Unless there is strong leadership support for the need to change, one or two individuals on a team can veto or otherwise obstruct improvement. It is a leadership responsibility to challenge teams to support new ways of doing things and reward and recognize teams that take the risks to use them.

The level of motivation for adopting change normally varies from team to team, from highly receptive to resistant so it is a bad idea to force teams to adopt a best practice. Teams forced to adopt a new practice are rarely effective at doing so. I recommend using a viral approach where pioneers

encourage teams to be the next set of adopters. When early adopters demonstrate results their initiative, risk-taking and results are publicly recognized. These teams then become promoters. The most credible promoter for a practice is a well-regarded engineer versus a senior manager or quality/process improvement leader.

6 Metrics Key to driving Effective Best Practice Adoption

Software metrics are a critical tool for driving effective engineering best practice adoption. Figure 4 below shows a key effectiveness metric for defect removal best practices – the number of minutes of effort to discover a defect by method. Defects discovered through personal reviews or peer reviews required an average of one hour of person-time to find and resolve, whereas the average defect found in system test required about eight to ten hours based on metrics I've seen from some teams at Adobe and Intuit.

It is best to use metrics that measure outcomes and not just activity. For example, having teams count the number of peer reviews they do is insufficient. Even if many peer reviews are being done if they are not finding defects then they are not accomplishing their purpose. What matters most are metrics like the percentage of defects removed using early defect removal methods. Ideally, defects are removed in the same phase they are injected when they are frequently an order of magnitude less costly than the next phase. Quality cannot be "tested into" a product.

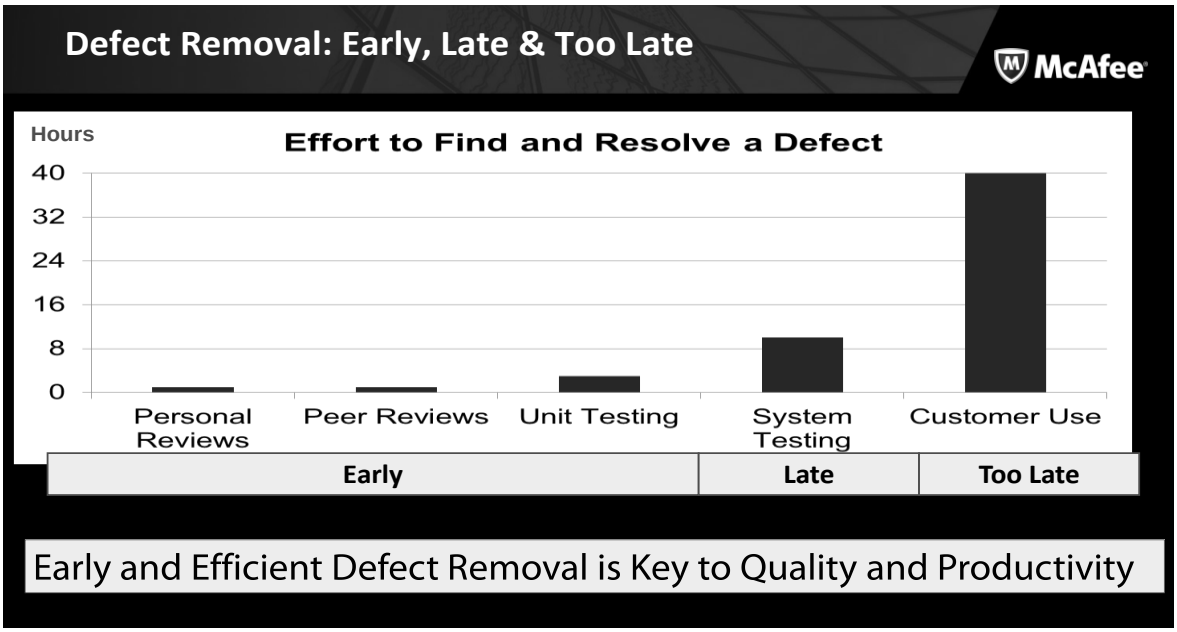


Figure 4: Defect Removal Effort by Method

The data from Figure 4 was collected from both McAfee and Adobe projects. You must measure the effectiveness of engineering practices to understand their return on investment and justify the necessary investments of time. Some teams tell me "Yes, we are already doing peer reviews". When I ask them how many defects they found, what the ROI was for that activity, and how many defects they are finding for each hour of time invested, some teams do not have answers.

To teach engineers the value of using software metrics, I ask them to review a real work product during a peer review workshop where we collect and use key metrics such as the amount of time invested and the number of operational defects found. Without personal experience collecting and

using software metrics to improve their own work, engineers often believe metrics are mainly for management.

I was approached by a team leader who said his team would be unable to participate in a Peer Review Workshop I was scheduled to teach because they lacked the time as his team was committed to deliver a software release to QA on the following day. After I mentioned that most teams find an average of three operational defects during the class he agreed to have his team participate. Ultimately, his team found four operational defects during their peer review, at least one of which would have certainly resulted in their missing a commitment to deliver a working software build the next day. After this experience the team leader and team agreed it was a good call to participate in the workshop and that peer reviews could be a very good investment of time.

7 Reducing Defect Driven Rework

A benefit of removing most defects prior to System Test is a significant reduction in engineering rework. Figure 5 shows the percentage of defects removed before System Test for a sample of seven Adobe TSP projects. TSP projects establish and manage goals and plans for removal of defects at each phase of the software engineering lifecycle. They also establish quality plans where they estimate the number of defects that will be injected and removed in each software development activity (e.g. design, coding) for each major component. These plans are used to assess whether a team is being efficient and effective in using early defect removal techniques.

Project	% of Defects Found Early	%Effort in Post-Dev Testing
A	**94%**	**11%**
B	**83%**	**15%**
C	**75 %**	**16%**
D	**78%**	**32%**
E	**88%**	**18%**
F	**83%**	**9%**
G	**75%**	**13%**
Average	**82%**	**16%**

Figure 5: Team Software Process Early Defect Removal Results

This reduction in post-release rework translated directly into these teams having 30% more time available for value-added work (see Figure 6). In this chart the total time to find and remove defects through all methods is compared (Total Cost of Quality) as well as the average pre-system test removal rate (% of Defects Found Early).

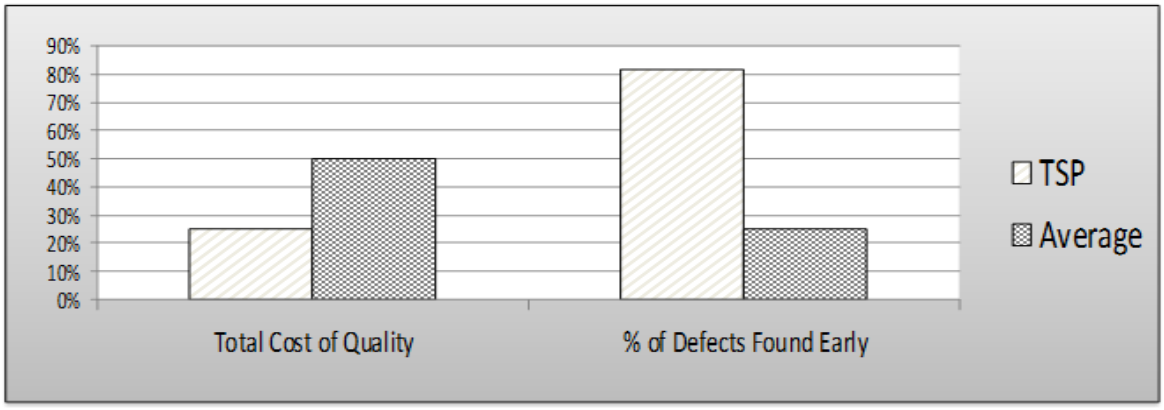


Figure 6: A comparison between TSP Projects and similar projects using traditional methodologies

These teams found a greater percentage of software defects prior to system testing. Defects found before system test are significantly less expensive to resolve. Also, the percentage of defects found before system test is directly correlated with post-release quality since System Testing will not find all defects. These teams were less frequently in fire-fighting mode and able to maintain more predictable work schedules and reasonable work-life balance. As a result, most of these teams had a larger portion of their development cycle available and allocated to design and implementation, versus being spent in a code-and-fix during system testing.

8 Improved Quality through Early Defect Removal

I've done studies at several companies that correlated the number of defects deferred for repair post-release with the number of defects discovered in System (QA) testing. The correlation between these two numbers ranged between .7 and .9. In other words, you can predict the number of defects that a project team will elect to not fix before production release (defer) if you know the number of defects found in System Testing. The two major reasons for the high correlation between these two numbers are:

1. When defects are found during QA testing they are relatively expensive to resolve. As a consequence, teams simply run out of time to fix all of the defects they find.
2. When defects are found in QA they are usually found late in the development cycle. Therefore, they are relatively more risky to fix and some are deferred to avoid the risk of introducing a more serious issue as part of a regression problem.

The bottom line is that an excellent way to increase both quality and engineering productivity is to find and remove a greater percentage of defects early.

9 Summary and Conclusions

To deliver quality work and improve how it is done, the senior leadership of the organization must expect and encourage it. Senior leadership support includes ensuring managers have quality improvement as part of their performance goals as well as regularly review progress with the teams. These quality reviews are also an opportunity to ensure quality is a priority and that the necessary time, dollars and headcount for engineering quality into the software are available and used. They also serve as a mechanism for identifying and spreading adoption of best practices across an organization.

A key enabler of quality is the adoption of engineering best practices including Peer Review, Scrum, Team Software Process and Unit Testing. Self-directed teams adopting TSP and/or Scrum are able to ensure that quality, schedule, scope and cost were not strict trade-offs. Through a combination of better planning and estimation, improved project execution and effective use of quality best practices such as unit testing and peer reviews, teams are able to deliver high quality software, on schedule and budget, with good work-life balance for the team. Learnings from retrospectives and improved metrics helped drive continuous and significant improvement in product and process.

Acknowledgements

I would like to thank Watts Humphrey and Noopur Davis for the transformative influence they have had on many of the teams that they worked with at Adobe and Intuit. I've learned so much about software engineering best practices and how to make this stick from Watts and Noopur. I would also like to thank Senior Leaders I've worked for at McAfee, Adobe Systems and Intuit including Bryan Barney, Barry Hills, Miles Lewitt and Bill Ihrie for their support.

References

1. Campanella 1999, *Principles of Quality Costs: Principles, Implementation and Use*, ASQ Quality Press.
2. Davis, N. 2003, *Team Software Process (TSP) in Practice*, SEI Technical Report CMU/SEI-2003-TR-014 (September 2003), SEI.
3. Humphrey, W. 2001, *Winning with Software*, Addison Wesley Professional.
4. Jones, C. 2010, *Software Engineering Best Practices*, McGraw Hill
5. Kotter, J. 1996, *Leading Change*, Harvard Business School Press.
6. Moore, G. 1991, *Crossing the Chasm*, Harper Business.
7. Reichfield F. 2006, *The Ultimate Question: Driving Good Profits and True Growth,* Harvard Business School Press*. and* http://www.netpromoter.com

Software Technology Readiness for the Smart Grid

Cristina Tugurlan, Harold Kirkham, David Chassin
Pacific Northwest National Laboratory
Advanced Power and Energy Systems
Richland, WA

cristina.tugurlan@pnnl.gov, harold.kirkham@pnnl.gov, david.chassin@pnnl.gov

Abstract

Budget and schedule overruns in product development due to the use of immature technologies constitute an important matter for program managers. Moreover, unexpected lack of technology maturity is also a problem for buyers. Both managers and buyers would benefit from an unbiased measure of technology maturity. This paper presents the use of a software maturity metric called Technology Readiness Level (TRL), in the milieu of the smart grid. (The smart grid adds increasing levels of communication and control to the electricity grid.) For most of the time they have been in existence, power utilities have been protected monopolies, guaranteed a return on investment on anything they could justify adding to the rate base. Such a situation did not encourage innovation, and instead led to widespread risk-avoidance behavior in many utilities. The situation changed at the end of the last century, with a series of regulatory measures, beginning with the Public Utility Regulatory Policy Act of 1978. However, some bad experiences have actually served to strengthen the resistance to innovation by some utilities. Some aspects of the smart grid, such as the addition of computer-based control to the power system, face an uphill battle. Consequently, the addition of TRLs to the decision-making process for smart grid power-system projects might lead to an environment of more confident adoption.

Biography

Cristina Tugurlan *joined Pacific Northwest National Laboratory as a software test engineer in August 2010. Before joining PNNL, she held a software engineer position at IBM, OR. She is responsible for the software testing and validation of projects modeling wind generation integration, power system operation and smart grid. She has a Ph.D. in Applied Mathematics from Louisiana State University, Baton Rouge, LA.*

Harold Kirkham *received the PhD degree from Drexel University, Philadelphia, PA, in 1973 and joined American Electric Power, and was responsible for the instrumentation at the Ultra-High Voltage research station. He was at the Jet Propulsion Laboratory (JPL), Pasadena, CA, from 1979 until 2009, in a variety of positions. In 2009 he joined the Pacific Northwest National Laboratory, where he is now engaged in research on power systems. His research interests include both power and measurements.*

David Chassin *has been at Pacific Northwest National Laboratory for 19 years and has more than 25 years of experience in the research and development of computer applications software for the architecture, engineering and construction industry. His research focuses on non-linear system dynamics, high-performance simulation and modeling of energy systems, controls, and diagnostics. He is the principle investigator and project manager of DOE's SmartGrid simulation environment, called GridLAB-D and was the architect of the Olympic Peninsula SmartGrid Demonstration's real-time pricing system.*

1. Overview of Paper

The paper begins by explaining how regulation of the power utilities led to a lag in the implementation of distributed automatic control. Re-regulation has changed the situation. The paper points out the need for assurance that any control now being introduced is ready for application.

The second section of the paper describes how Technology Readiness Levels (TRLs) were introduced and used by different agencies and institutions in their technology development programs.

The third part of the paper develops a variant of the TRL system that is relevant to the smart grid, and in particular to the software that must be employed. The example of GridLAB-D is presented. The relationship between TRL and testing is discussed.

Finally, it is argued that the TRL sequence leaves a trail of evidence that can be used to overcome a well-established risk aversion. The paper should thus be of value to implementers of the smart grid, as a way to become more confident of the software status.

2. The Smart Grid

In economics, a “natural” monopoly can be defined as an “industry in which multiform production is more costly than production by a monopoly” (Baumol, 1977). The situation can arise because the fixed cost of the capital goods is so high that an economy of scale can be arranged so that it is not profitable for a smaller second firm to enter and compete. To prevent utilities from exploiting their monopolies with high prices, they have typically been regulated by government. Utilities such as water, electricity, and natural gas used to be protected monopolies, guaranteed a certain rate of return on their investment. As a result they developed entrenched risk-avoidance behavior, and resistance to innovation. However, the notion that the power companies were a natural monopoly began to be questioned in the 1970s (The Economist, 1998). The idea that there were limitless economies of scale was thought no longer valid, and the point was made law by the Public Utilities Regulatory Policy Act of 1978, that *required* power companies to buy energy from qualifying competing facilities.

Nowadays, electricity has undergone a period of deregulation, and the generators of electric power can now compete. But the infrastructure, i.e. the wires that carry the electricity, usually remains a natural monopoly, and the various companies send their electricity through the same grid. To accommodate a number of new kinds of generation, a *smart grid* is being created. The smart grid essentially takes an electricity grid and intelligently integrates communications and computer technology, so that (for example) suppliers can deliver electricity to consumers in a wider range of conditions, while also accommodating wind and solar power sources.

Thus, utilities in the power sector deal with a shift towards smart grid and energy efficiency. Increasingly, smart grid projects include software-based efforts. Compared to hardware endeavors, a major advantage of this type of projects is a shorter time to market.

Just as with hardware, there is a real need for a metric to measure the technology maturity and integration level of the software in the smart grid power system projects. A scale called Technology Readiness Levels (TRLs), adapted from other fields, is proposed. Because of the documentation required, the metric can be used to increase the comfort-level of utilities as they adopt the new directions, and take advantage of commercialization.

2. Technology Readiness Level and Software Quality

In any project there are multiple interdependent constraints that influence the project success. Three crucial ones are scope, time, and budget. However, many projects somehow get started without sufficient previous planning, and the work-load seems to increase unexpectedly as the projects progress.

According to Donna Shirley (2010), manager of the Mars program at the Jet Propulsion Laboratory, the business of TRLs got started at National Aeronautics and Space Administration (NASA) because of a "guy named Werner Gruel, who was a NASA cost analyst." Gruel "had this great curve that showed that if you spent less than 5 percent of the project cost before you made a commitment to the cost, that it would overrun. If you spent 10 percent of the cost before, it wouldn't overrun." Figure 1 is an updated version of Gruel's graph, showing the overruns on some actual space missions.

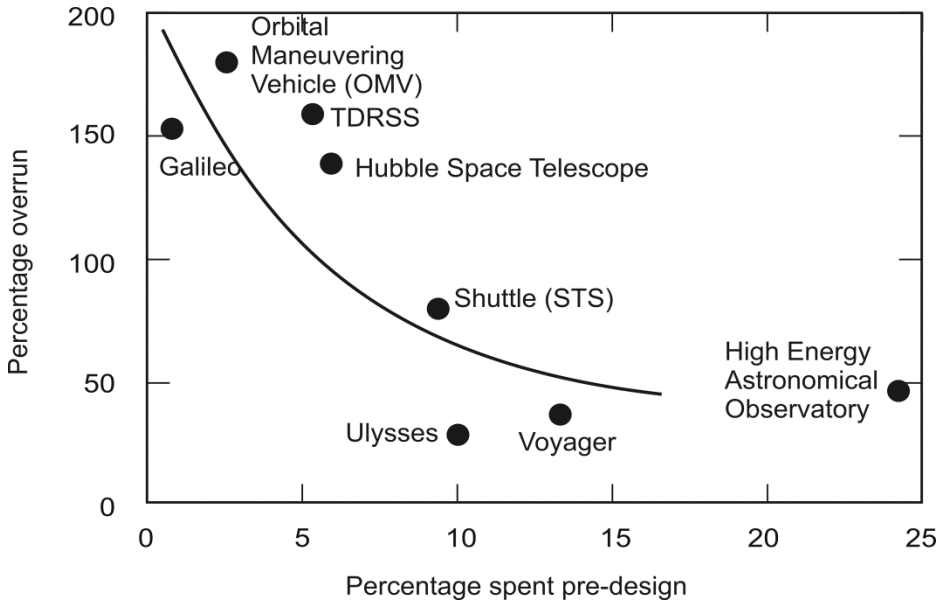


Figure 1: Cost overrun as a function of pre-design spending

What was going on, according to Ms. Shirley, was that at the start of a project, there was a need for a cost estimate, so the engineers did a quick-and-dirty design that they called a "point design." A cost estimate based on the point design was then put into the rather long NASA budget cycle.

However, the chances were good that the point design did not take everything into account. After the funding has been obtained, and work had started, it was gradually realized that there was going to be a cost overrun. At this point, the choices were between an overrun and a descope. Nobody wants to descope, so the result was usually that there was a cost overrun, and the engineers got the blame. What NASA needed was a way to improve their understanding of what was involved. This is where the Technology Readiness Levels came into their own.

TRLs had been under development at NASA since the 1970s, and the curve generated by Werner Gruel provided the ammunition to show they could be useful. The original TRL concept came from Stan Sadin at NASA, in 1974, with seven levels identified (Sadin 1974). The method was developed sporadically after that. The current nine-level version of TRLs was written down in a white paper by John Mankins at NASA in 1995, and seems to have become more or less fixed. Since then, the European Space Agency has adopted something very similar. So has the US military. (The US Air Force has even created a spreadsheet to help determine the TRL of technology. So has the Department of Homeland Security.)

The TRL definitions have some variation in interpretation to suit different organization's needs, but their overall scales match NASA's traditional scale quite closely, focusing on how ready a

technology/integration/system is for its application in the intended environment. In Table 1, Weiping and colleagues (Tan et al. 2009) compare the TRLs used by the leading government agencies in their technology development programs. The agencies tailored the TRL process specifically to their organizations in order to produce operational systems on schedule and within budget. The NASA TRLs are the ones listed by Mankins (Mankins 1995), and the Department of Defense (DoD) ones are from the DoD Deskbook (DoD 2005). The Department of Energy (DOE) TRLs here apply only to the nuclear fuel side of DOE (Carmack and Pasamehmetoglu 2008). NATO TRLs are given in a 2006 document (NATO, 2006).

Table 1: TRL definitions used by the leading government agencies

TRL	National Aeronautics and Space Administration (NASA)	Department of Defense (DOD)	Department of Energy (DoE)	North Atlantic Treaty Organization (NATO)
0	N/A	N/A	N/A	Basic research with future military capability in mind
1	Basic principles observed and reported	Basic principles observed and reported	Initial concept verified against first principles and evaluation criteria defined	Basic principles observed and reported in context of a military capability shortfall
2	Technology concept and/or application formulated	Technology concept and/or application formulated	Technical options evaluated and parametric ranges are defined for design	Technology concept and/or application formulated
3	Analytical and experimental critical function and/or characteristic proof-of-concept	Analytical and experimental critical function and/or characteristic proof of concept	Success criteria and technical specifications are defined as a range	Analytical and experimental critical function and/or characteristic proof of concept
4	Component and/or breadboard validation in laboratory environment	Component and/or breadboard validation in laboratory environment	Fuel design parameters and features defined	Component and/or breadboard validation in laboratory/field (eg ocean) environment
5	Component and/or breadboard validation in relevant environment	Component and/or breadboard validation in relevant environment	Process parameters defined	Component and/or breadboard validation in a relevant (operating) environment
6	System/subsystem model or prototype demonstration in a relevant environment (ground or space)	System/subsystem model or prototype demonstration in a relevant environment	Fuel safety basis established	System/subsystem model or prototype demonstration in a realistic (operating) environment or context
7	System prototype demonstration in a space environment	System prototype demonstration in an operational environment	All quantification steps completed and fuel is licensed	System prototype demonstration in an operational environment or context (eg exercise)
8	Actual system completed and “flight qualified” through test and demonstration (ground or space)	Actual system completed and “flight qualified” through test and demonstration	Reactor full-core conversion to new licensed fuel completed	Actual system completed and qualified through test and demonstration
9	Actual system “flight proven” through successful mission operations	Actual system “flight proven” through successful mission operations	Routine operations with licensed fuel established	Actual system operationally proven through successful mission operations

When first invented or conceptualized, most new technologies are not suitable for immediate application. They must go through a number of stages including experimentation, refinement, and extensive testing, in order to be proven ready for system integration and/or commercialization. As the TRL number gets higher, the range of applications gets narrower. At very low TRL numbers, there could be many applications for a particular technology. TRL 2 is not reached until there is one application in mind. As the TRL number gets higher, the cost of moving from one level to the next gets higher. This cost increase is brought about by the need to involve more specialized workers, and the need for keeping track of things with more documentation.

Figure 2, adapted from Nolte's "whale-shaped" chart (Nolte 2008), shows how the usefulness of a particular technology evolves in time, and how the technology development phases relate to the TRLs. It is clear that most of the TRLs occur early in the technology life cycle, where:

- TRL 1 to TRL 3 address general conceptual science matters,
    - a step up from TRL1 to TRL2 shifts ideas from pure to applied research,
- TRL 4 to TRL 5 cover the transition from scientific research to engineering and system development,
    - TRL 4 is the first step in determining whether the individual components will work together as a system (DOE 2009),
- TRL 6 to TRL 9 focus on engineering matters,
    - TRL 6 begins true engineering development of the technology as an operational system,
    - TRL 9 represents the final stage of the technology, when the technology is fully operational and its maturity is reached.

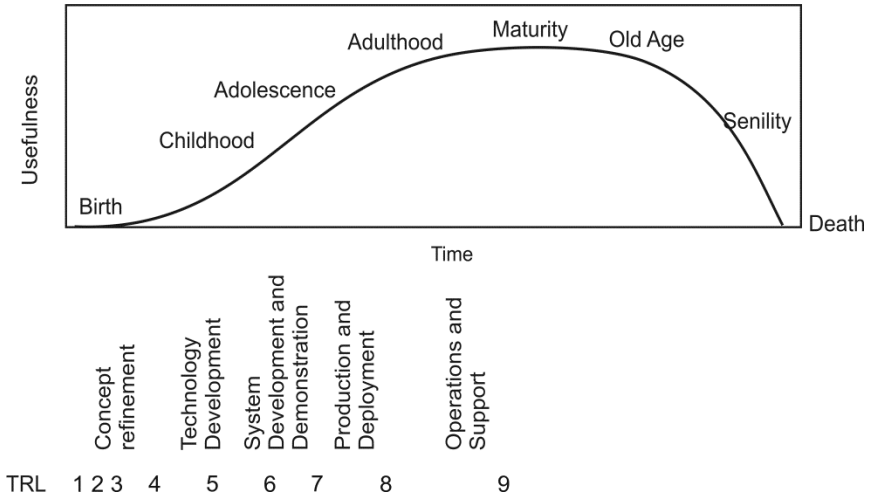


Figure 2: Technology Life Cycle and Technology Readiness Levels

3. Software Technology Readiness Level in Smart Grid

The TRL metrics are applicable to any type of technology, including hardware and software products. Measuring the readiness of a software product reflects some combination of quality characteristics estimated at a given moment of time. However, it is impossible to produce systems of any size which do not change as they develop, so the TRL number is a transient thing. Both the hardware and software environments surrounding a software product change and therefore the software products are continuously changing and aging (Eick et al. 2001).

Once software is put into use, new requirements emerge and existing requirements change, for example as the business running the software changes. Parts of the software may have to be modified to support architectural changes or to correct errors that are found in operation, improve its performance, or other non-functional characteristics. Consequently, even after delivery, software systems evolve in response to demands for change.

Building the smart grid requires a lot of computer and software subsystems. Utilities begin to address the strategy, management, and regulatory issues involved in transitioning to a smart grid. Consequently, the utilities might benefit from using TRLs in their technology acquisition processes.

For hardware, implementing TRLs for the smart grid is not a difficult thing to visualize. The manufacturer of a widget intended for the smart grid can assemble convincing evidence during the design and development process that the widget is at a level of development that it would not endanger the power system if it were put into the field. It has to be admitted that, in the past, due to the lack of this kind of documentation, some technologies have been applied to the electric power system before they were really ready. For example, back in the 1960s, some early applications of solid state technology to power system relaying were less than adequate. Had the devices been subject to the rigor of an appropriate qualification program, that experience could have been avoided. However, because the utilities did have this bad experience, future technology diffusion has to make the effort to be convincing. Hasty adoption of unready technology can sometimes lead to delays in the adoption of ready technology.

For software, the case of an application developed at Pacific Northwest National Laboratory (PNNL) is considered. To assure the correct functionality of some products used in the distribution part of the smart grid, a computer model simulator, called GridLAB-D, was developed at the U.S. Department of Energy's Pacific Northwest National Laboratory. The sophisticated computer program provides a detailed, simultaneous simulation of the electric grid, including power flow, end-use loads, and market functions. In other words, it models both technical and non-technical interactions within a power grid. The model allows users to evaluate new technologies and operational strategies, to craft and refine the characteristics of these technologies and strategies, and to predict the results of deploying them. GridLAB-D was designed as an open source system having contributors all over the world. Modeling in GridLAB-D uses specific classes of objects, and modules, classified by functionality.

The GridLAB-D project team began using TRLs in January 2011 to get a better understanding of technology status, manage risks and make decisions in funding, development and deployment. Moreover, TRLs assist with assessing the readiness of modules and classes for analysis projects and studies. The software TRLs, tailored for the GridLAB-D project, are detailed in Table 2. To assist with an effective and consistent use of TRLs in Table 2, a methodology of software readiness assessment is designed. Unfortunately, the procedural steps and the evaluation of the obtained results are not yet in final form, since the design process is constantly optimized and improved.

Table 2: Software Technology Readiness Level

TRL	Definition	Description
1.	**Basic principles described** (mathematical formulation)	Lowest level of software readiness. Basic concept begins to be translated into applied research and development by providing a detailed mathematical formulation. Examples might include a concept that can be implemented in software, or analytic studies of an algorithm's basic properties.
2.	**Application concept formulated** (algorithm)	Development has begun. Basic individual algorithms or functions are prototyped and documented. Results are speculative and there is no proof or detailed analysis to support assumptions or expectations. Examples are still limited to paper studies.
3.	**Analytical proof of concept** (prototype)	Active research, development and documentation are initiated. Depending on the size and complexity of the implementation, basic components of the integrated critical system have been designed, built and partially tested. Analytic studies to produce code that validates analytical predictions of separate software elements of the technology are done. Examples include implementation of software components that are not yet integrated or thoroughly tested, but satisfy an operational need. Algorithms are run and tested on a surrogate processor in a lab environment.

4.	**Standalone component validated** (earliest version)	Basic software components are integrated to establish that they will work together. They are relatively primitive with regard to efficiency and reliability compared to the eventual system. System software architecture development initiated to include interoperability, reliability, maintainability, extensibility, scalability, and cyber-security issues. Software integrated with simulated current/legacy elements as appropriate. Verification and validation process is partially completed, or completed for only a subset of the functionality in a representative simulated laboratory environment. Documentation includes design documents and a start of a user manual.
5.	**Integrated component validated** (ALPHA version)	Reliability of software ensemble increases significantly. All software components are integrated with reasonably realistic supporting elements so that the software can be tested and completely validated in a simulated environment. Examples include "high fidelity" laboratory integration of software components. System software architecture is established. Algorithms run on a processor(s) with characteristics expected in the operational environment. Software releases are "Alpha" versions and configuration version control is initiated. Full documentation according to the applicable software standards, test plans and application examples, including all use cases, cyber-security and error handling should be provided.
6.	**System - subsystem demonstrated** (BETA version)	Represents a step up from lab scale to engineering scale. Representative model (BETA version), which is well beyond that of TRL 5, is tested in a relevant environment. Examples include testing a prototype in a live/virtual experiment or in a simulated operational environment. Algorithms running on the simulated operational environment are integrated with actual external entities. Configuration control and quality assurance processes are fully deployed. Verification and Validation process is completed for the intended scope (including robustness) and the system is validated in an end-to-end fully representative operational environment (including real target).
7.	**Prototype demonstrated** (product RELEASE)	Requires the demonstration of an actual system prototype in an operational environment. Algorithms running on processor of the operational environment are integrated with actual external entities. Software support structure is in place. Software releases are in distinct versions. Functionality and performance are not significantly degraded by frequency and severity of software deficiency reports. Verification and validation is completed, validity of solution is confirmed within intended application. Requirements specification are validated by the users. Engineering support and maintenance organization, including helpdesk, are in place.
8.	**System "analysis qualified"** (general product)	Software has been demonstrated to work in its final form and under expected conditions. In most cases, this TRL represents the end of system development. Examples include test and evaluation of the software in its intended system to determine if it meets design specifications. Software releases are production versions and configuration controlled, in a secure environment. Software deficiencies are rapidly resolved through support infrastructure. Full documentation including specifications, design definition and justification, verification and validation (qualification file), users and installation manuals, training and education materials, software problem reports and non-compliances should be provided.
9.	**System proven** (live product)	Represents actual application of the software in its final form and under designed conditions, such as those encountered in operational test and evaluation. In almost all cases, this is the end of the last "bug fixing" aspects of the system development. Examples include using the system under operational design conditions. Software releases are production versions and configuration controlled. Frequency and severity of software deficiencies are at a minimum. Sustaining engineering, including maintenance and upgrades, updates to documentation and qualification files are in place.

The assessment of software readiness level required individual estimations and evaluations (a subject matter expert assesses the TRL of the technology). This is followed by group estimations in meeting or conference format for discussing the technology maturity, and a combination of the above when a

consensus of a single estimate is sought. For each GridLAB-D class, the frequency distribution of the individual estimations and evaluation is constructed and incorporated in the mean value calculation. The method of determining TRL of a class is difficult to be extended to modules because classes can often interact in ways that are not measured by the class TRL. The TRL of each GridLAB-D module is calculated as the lowest TRL found in the classes used by the model. Since the interaction between classes is not quantized in the TRL grading, the actual TRL of any model may be lower than the TRL computed by GridLAB-D. For illustration purposes, Table 3 presents the TRLs assessment for two GridLAB_D models - climate and residential. By choosing what classes are included in a specific model, the user has the possibility to increase or decrease the TRL of that model, i.e. a *residential* home having *house_e* and *water heater* as components has a modul TRL of 7, while *house_e* .by itself has a module TRL of 9.

Table 3: GridLAB-D TRL assessment

Module	**Classes**		**Module TRL**
	Name	**TRL**	
climate	*weather*	*9*	*9*
residential	*clothes washer*	*2*	*2*
	dishwasher	*2*	
	dryer	*2*	
	evcharger	*4*	
	freezer	*5*	
	house_a	*6*	
	house_e	*9*	
	lights	*8*	
	microwave	*5*	
	occupant load	*8*	
	plug load	*8*	
	range	*5*	
	refrigerator	*5*	
	water heater	*7*	
	zipload	*9*	

All of the above proved that TRLs can result in more reporting, paperwork and review time. It takes time for participants in GridLAB-D work to adjust to using TRLs and for the TRL process to have an effect project-wide. Understanding and communicating the detail of the TRL assessment is vital to avoid unnecessary concerns. For example, some elements may have a low TRL but a clear development path and therefore their maturation is a low risk.

Systems engineering processes are not addressed in the lower TRLs, which can result in difficulties transitioning technologies to higher TRLs. There are situations when moving a TRL up on the scale is a matter of providing resources to rapidly increase the TRL or seek an alternative solution (technology) with a higher TRL.

It may be noted that the TRL scale of Table 2 does not exactly parallel the TRL values in Table 1. For example, in the (hardware-based) Table 1, TRL 4 and 5 are no more than breadboard systems, and it is sometimes said that one can reach TRL 6 in a good Hobby Shop. (Mind you, TRL 5 at NASA requires that a failure modes and effects analysis (FMEA) is performed, not something usually associated with a Hobby Shop.) In contrast, TRL 5 in Table 2 talks about “releasing” software, a process that surely involves leaving the laboratory or workshop environment. This difference in process results from the difference in testing needs between hardware and software.

Based on the TRL, stating how successful each subsystem of the hierarchy will be requires that the TRLs be linked to test plans and trials. This linkage provides a clear statement on the TRL achievement at each stage of the project. The TRL definitions provide a convenient means to further understand the

relationship between the scale of testing, fidelity of testing system, and testing environment and the TRL (DOE 2009). As it can be seen in Table 4, the scale requires that for a TRL 6 testing should be completed at an engineering or pilot scale, with a testing system fidelity that is similar to the actual application. TRL 6 represents the point when the software is delivered to customers for beta testing. Therefore, the verification and validation process should be completed for the intended scope and the system should be validated in a simulated environment with some external features. The TRL scale used in Table 2 requires that testing of a prototypical design in a relevant environment be completed prior to incorporation of the technology into the final design of the facility.

The Technology Readiness Level is, of course, not just monitoring for the sake of monitoring. It is a useful part of the management process. TRL levels were originally intended as a way to assess the amount of effort needed to get something qualified for space use. Implicit in that statement is the fact that if a widget was not at a good solid TRL 6 or even a TRL 7, it was *not* going into space. Whether for manned-space or not, these levels were important. In adapting the idea to software, it is not unusual to release software to the users in a somewhat unprepared state. What this means is that at a TRL that would just about get a widget into space, the software is released to Beta test. Therefore, the important TRL number is really TRL 6. What is needed up to TRL 6 controls whether the software leaves the workshop. This is the message to take away from the TRL numbers in Table 2.

Table 4: Relationship of Testing Recommendations to the TRLs

TRL Level	Scale of testing [1]	System Fidelity [2]	Environment [3]
1		Paper	
2		Paper	
3	Lab	Pieces	Simulated
4	Lab	Pieces	Simulated
5	Lab/Bench	Similar	Simulated
6	Engineering/Pilot Scale	Similar	Relevant
7	Full	Similar	Operational (limited range)
8	Full	Identical	Operational (full range)
9	Full	Identical	Operational (full range)

Numbered notes

1. Scale of testing
- Full Scale matches final application.
- 1/10 Full Scale < Engineering/Pilot Scale < Full Scale (Typical)
- Lab Scale < 1/10 Full Scale (Typical)

2. System Fidelity
- Identical – matches final application in all respects
- Similar – matches final application in almost all respects
- Pieces – matches a piece or pieces of the final application
- Paper – exists on paper

3. Environment
- Operational (full range) – full range operational capacity
- Operational (limited range) – limited range operational capacity
- Relevant – simulated environment plus a limited range of external features
- Simulated – restrictive range of simulation

4. Conclusions

Properly applied in system engineering management, Technology Readiness Levels are of value in understanding technology status, managing risks and making decisions in funding. For the development and deployment of software products they assist the system engineer and the manager. The TRLs reduce the amount of subjectivity about project status, and therefore, planning and execution can be managed better.

TRLs can serve a number of useful purposes in the electric generation and delivery world. Overall, the transitioning of technology from laboratory to application becomes a carefully managed process. For the utility world and the hardware headed for the smart grid, the TRL sequence leaves a trail of evidence that can be used to overcome a well-established risk aversion.

The same observation is necessarily true for software. However, for software there are a few additional difficulties in applying the standard TRL method. For example, inevitable software "decay", architecture transformations and the need for software maintenance after release, increase the volume of documentation involved. Nevertheless, the documentation serves a purpose, and it can be argued that it justifies its existence.

The proposed software TRLs described in this paper have been applied to an actual software system development, i.e. GridLAB-D. While the application of TRLs to the software is presently in its relatively early stages, it strengthens the smart grid software capabilities even further by pinpointing and even preventing system malfunction before release to the customers.

References

DOD 2005 May. Technology Readiness Assessment (TRA) Deskbook,

DOE G 413.3-4 2009, October. US DOE Technology Readiness Assessment Guide,

The Economist 1998, March. Power to the people: Deregulation and new technology are working hand in hand to transform the global electricity-supply industry,

Eick, S., Graves, T., Karr, A., Marron, J., and Mockus,A. 2001. Does Code Decay? Assessing the Evidence from Change Management Data. IEEE Transactions on Software Engineering, Vol. 27, No. 1,

Mankins, John C. 1995, April. Technology Readiness Levels: A White Paper. NASA, Office of Space Access and Technology, Advanced Concepts Office,

NATO 2006. North Atlantic Treaty Organization (NATO) Technology Readiness Levels. http://www.nurc.nato.int/research/trl.htm,

Nolte, William L. 2008. Did I Ever Tell You about the Whale? or Measuring Technology Maturity. Information Age Publishing,
,

Sadin, Stan 1974. Origin of TRL. http://en.wikipedia.org/wiki/Technology_readiness_level
Shirley, Donna 2010. www.jsc.nasa.gov/history/oral_histories/NASA_HQ/.../DLS_7-17-01.pdf,

Tan, Weiping, Ramirez-Marquez, Jose, and Sauser, Brian 2009. A Probabilistic Approach to System Maturity Assessment. Wiley Online Library, DOI 10.1002/sys.20179.

Volunteer Armies Can Deliver Quality Too

Achieving a Successful Result in Open Source, Standards Organizations, and Other Volunteer Projects

Julie Fleischer
Julie.N.Fleischer@Intel.com

Abstract

Delivering quality can be straightforward in an organization that creates multiple products and has a series of time-tested processes and procedures in place to guarantee success. However, at some point in your career, you may be asked to deliver a quality result with a team that has self-defined or loosely-defined processes, includes volunteers whose "day job" is not your project, and who rely on a consensus process with individuals from diverse, and sometimes competing, backgrounds and agendas to make decisions. This paper is written by a Program Manager and QA Lead who has spent the better part of the past decade delivering quality test and certification programs, specifications, and projects in various open source programs and standards development organizations. We'll cover tips and techniques for achieving success in those volunteer-based environments.

Biography

Julie Fleischer has worked at Intel for the past thirteen years and has held a variety of roles in program/project management, QA leadership, and software development. She currently is a Technical Program Manager for Linux driver development and also briefly held a position as the Technical Program Manager for the Yocto Project (www.yoctoproject.org), an open source project for creating embedded Linux distributions. Prior to those roles, she worked as the Chair of the Test and Certification Working Group for Continua Health Alliance where she led the team to define and implement a certification program towards v1.0 and v1.5 of the Continua Design Guidelines. She also chaired the team that created the USB Personal Healthcare Device Class standard and project managed the creation of the v1.0 Continua Design Guidelines. She has spent over a decade in open source and standards organizations and has led many "volunteer armies" to accomplish successful results.

Julie received her BS and MS in Computer Science from Case Western Reserve University and has presented at PNSQC in the past on technical and interpersonal topics.

1. Introduction

What is a volunteer army?

A volunteer army is a team of more than one person where at least some members of the team are not getting financially compensated for the work they do. In other words, their "day job" may not be to work on the project that the volunteer army is working on. These individuals typically come from diverse, and sometimes competing, backgrounds and agendas. In the software world, volunteer armies can be found in open source projects, standards organizations, or other special interest groups. These volunteer armies may not have formal quality assurance teams or plans and may not have official project management roles or practices.

Decision making in a volunteer army is typically consensus-based; however, depending on the parent organization, decision making can be hierarchical or even a mix of types. For example, a standards organization may have strict voting rules on requirements for voting eligibility and participation, but an open source project may use meritocracy and hierarchy in order to make decisions affecting the project.

Not all processes in a volunteer army may be explicit or documented. Standards organizations may have clear membership and voting rules as well as requirements on publication of standards; however, they may not have requirements on how meetings are run, how often members meet, and how day-to-day decisions are made. Depending on the size and complexity of an open source project, it may or may not have a website or Wiki defining the processes and procedures used for submitting and triaging bugs or developing and releasing code. However, processes describing how decisions are made and how the group is managed may be lacking.

Delivering quality products in environments like these can pose unique challenges. It can be difficult to get a group of part-time participants with competing agendas to come to consensus. It can also be challenging to make progress when the team members have unpredictable schedules and availability. Further, maintaining motivation with a team that has competing priorities and interests requires a special finesse. This paper will describe tips for achieving quality results in volunteer armies like these along with anecdotes and advice to assure you that volunteer armies can deliver quality – and have fun while doing it!

2. Respect your volunteer army.

Perhaps the most effective best practice for making progress and delivering quality results in a volunteer army is to respect the volunteer army. Since volunteer armies aren't financially compensated for their work, they require a sense that they are adding value and significant contributions to feel that their time is well spent in a volunteer organization. Eric S. Raymond calls this feeling "egoboo."[i] That is, the sense of enhancing one's reputation among others in the group. As a leader, or active participant, of a volunteer army, it is important to show respect for your volunteer army. This can be anything from simple "Thanks!" to more formal celebrations and parties when warranted.

Some techniques that I have found effective are:

1) A "Thank you!" goes a long way. I like to make my "Thank you"s explicit and let the person know exactly what they did that was worthwhile. This helps encourage the same kind of activity again since the individual knows what was valued.
2) Listening and letting others talk is the best way to encourage more feedback. When you don't interrupt others, allow conversations to take their time, and actively document and respond to other's input, it tells them their opinions are worth-while and valued.
3) Just because a volunteer organization may not have the budget to have the formal milestone parties that a company does doesn't mean you can't celebrate milestones. When a team I chaired finished writing a technical specification, we all knew we wouldn't be meeting again face-to-face to celebrate. So, we held a "virtual" celebration, complete with awards, virtual beer (i.e. a

power point picture of a pint of beer - The team didn't find this as satisfying as the real thing, but appreciated the gesture.), and sharing anecdotes from our work together.

Although listening and letting others express their ideas is one way to show respect for your volunteer army, it is still important to guide discussions. Volunteer organizations often can get new members with many ideas, which may or may not be in the direction of the group. Standards organizations frequently face this issue when a face-to-face meeting may bring out many different attendees than the attendees on regular con-calls. While it is important to solicit and listen to newcomers ideas in order to ensure the newcomers become regulars and feel valued, sometimes you also need to ensure the group is true to their original vision. It is important to remember that listening to others and respecting their opinions does not have to mean changing your original plan. Taking the time to listen to a new idea and investigate it if appropriate is often enough to make a newcomer feel their opinion is valued. Most people don't expect that every idea they provide will be acted on and implemented, but they do appreciate a forum where their ideas are listened to and considered. If a conversation appears to be going on too long, taking a follow up action item to investigate is often an effective means for ensuring opinions are heard and the team stays on track. In addition, section 5 will talk about one technique for maintaining direction in a conversation by using a straw man proposal.

3. Always have a list of tasks needing volunteers

One quick way to ensure you can engage volunteers from the start is to make your "to do" list for your project or team visible to any volunteer that might want to join. This can take many forms: Wiki page, bug database, web site, Kanban, document, or anything else that is effective for the project. It is useful if your task list contains an idea of the scope of the task in terms of time required and technical expertise and experience required to complete. If you have the time, a phone call to a prospective new member to introduce them to your project as well as review the requirements of the tasks that that member may be a fit for can help integrate that member quickly.

4. Don't be afraid to jump in.

In organizations with clearly defined roles and responsibilities, a program manager, QA lead, or other person responsible for making things happen, has a management escalation path to ensure the project is getting the maximum effort and prioritization from the individuals on the team. However, with collections of volunteers, those lines of authority don't exist. If you want to encourage others to participate, you sometimes need to take the initiative yourself. This means being willing to do whatever is required to make sure the team makes progress, even if it's not something that is exactly in your skill set.

Some ways I have used this tip successfully in the past were:

1) When I chaired a working group which was writing a specification for transport connectivity of consumer electronic devices and personal health devices, I relied on one expert and other technical and personal healthcare experts on the team to establish our architecture and design. However, these people didn't have time to write the specification. So, I took on the job of translating the architectural decisions that we agreed upon in meetings into specification language. With the additional help of a great technical editor, we were able to create a readable and implementable specification.
2) When someone sends out a document or proposal for review in a volunteer group and has disclosed to me that they want the group's feedback, I like to use the "Reply All" technique on my review to encourage others to jump in and perform a review as well. I especially find this effective if I add in questions to others or describe areas where someone else's expertise may be of assistance. This encourages them to look at the area I called out and maybe perform an even larger review.

In those cases when you are not a leader but simply a participant, the only way to become a part of the group may be to simply jump in. This is especially true in groups where there is no clear leader or

defined roles and responsibilities, such as in school or extra-curricular volunteer organizations. This technique is best used when combined with other techniques, such as respecting your volunteer army (see section 2) to ensure you take on a reasonable amount but still empower others to take on roles as well.

5. Begin decision making with a straw man proposal.

There is an old folk story about a group of hungry travelers who would like food from the local villagers, but have no compensation to offer[ii]. As the story goes, the hungry travelers begin by filling a pot with water and a stone and declare that they will make "stone soup." The amused villagers watch in amazement as the travelers begin heating the pot to cook their "stone soup." When one traveler acknowledges that a little salt is needed to give the soup taste, a villager agrees to provide just a pinch of salt to the cause. Next, the travelers ask for barley to give it texture, and another villager steps forward. Then, they ask for and receive carrots to give it flavor. Each villager figures they can part with just a small amount of food or seasoning since they don't have to give everything in their larders. By the end of the story, the pot is filled with a delicious soup that all the travelers and the villagers enjoy.

This folk story about making something from simply a solid ("stone") base illustrates how people can more easily be motivated to provide the valuable resource of their input and energy once they understand they don't have to be responsible for the success or completion of the project as a whole. Once the villagers realized that they didn't have to cook an entire meal for the travelers, just provide some carrots or seasoning, they were much more willing to step forward and offer input. In technical groups, the "stone" is typically called a straw man proposal. This is a proposal may be nearly complete except for corner cases or may be as basic as simply a framework for encouraging team input.

I find a straw-man proposal is quite effective in discussing technical problems. With a basic straw man proposal, the team's work becomes that of finding corner cases and fleshing out the proposal, opposed to brainstorming a solution on the spot. Sometimes, the technical problem has multiple potential solutions and the team needs to choose between them. In this case, creating multiple straw man proposals and pros/cons lists can help the team better debate the merits of the proposals and possibly even create a new proposal which leverages the pros of the strongest proposals.

I have also found it effective to use a straw man proposal as a mechanism for making progress in a variety of other situations where I know I may not get all the feedback I want during a meeting. For example, if the team needs to brainstorm the agenda for our next face-to-face, I will typically lay the framework, placing in our regular discussion topics and our lunch and other breaks. After seeing the groundwork, the group is always able to fill in the open spaces with important topics I could not have come up with on my own.

6. Work within the system first, then propose change

As a newcomer to a volunteer army, it is essential to your success to understand the existing system and processes before you work to promote changes. It is challenging as a newcomer to come up to speed on the processes and procedures of the group you are joining, and there is a strong temptation (sometimes subconscious) to encourage the team you are joining to abandon their current processes in favor of a process that you are familiar with and that worked well in environments you were in previously. However, it is important to understand that the team you are joining, especially a team of volunteers, will have a high resistance to change unless they fully understand your proposal and agree that the pay offs are significant.

I encountered this situation when I was hired as the program manager for an open source project that had been running successfully before I came on board. I was able to implement some changes to the

process because I waited to implement them until I understood the full environment. For example, the team had solid processes in place for organizing and tracking the progress of their work, and I spent my first few months understanding and adopting these. However, I noticed that something was occurring in those months. A few weeks after I joined, I asked the question, "Did we finish milestone X?" And, a flurry of emails were sent so that the team could officially package up milestone X and announce that milestone X was completed. A month or so later, a few members of the team noticed that milestone X+1 had come and gone without any packaging or announcement. It was at this point that I realized that the processes that had been working well for the team in tracking progress were not working in officially releasing software. Typically, milestones were passed without enough fanfare for the team to even notice the passing. Because this was a need that the team agreed existed and because I had built up a rapport adopting the other processes of the team, I was able to add in the concept of release criteria and hold release readiness meetings before all subsequent milestones to ensure that the milestone criteria were met and the milestone was officially released and announced. Had I come in during my first few weeks and attempted to propose release criteria, I believe the team would have strongly resisted the "extra process and paperwork" I was adding. However, because I waited until I understood the team and the team trusted me, I was able to make this change more easily.

When you attempt to understand the environment you are entering first and then begin proposing changes, you are much more likely to achieve success. This fact is especially true in volunteer organizations, such as the open source project I was part of, because the group is already together because of a common passion and a pride in the work they do. I have been in volunteer organizations where a newcomer will come in with feedback on how to "fix" our "broken processes" and "solve" all of our problems without really understanding our true problems or processes needing support. When this happens, the newcomer typically finds resistance much stronger than they expected and, without an understanding leader who follows the tips above, the newcomer may eventually leave in frustration.

7. Focus on the need, not the name.

Once you understand the system and documented or undocumented processes that your volunteer army is following and are interested in motivating the group to change, it is useful to focus on the need that your idea is solving, opposed to focusing on championing a specific technique or process you knew worked in a previous organization, especially a previous non-volunteer organization. Especially in volunteer armies that pride themselves on being volunteer armies, such as open source projects, attempting to bring in process and procedures, and especially jargon, from non-volunteer armies, such as closed source projects, can backfire.

When I was a newcomer to an open source project, I was actually asked to do exactly what I just advocated not doing. I was asked to create a "Product Requirements Document" by some members of the team, despite the fact that formal "Product Requirements Documents" are typically not created in open source projects.

Initially, I followed this advice and began creating a Product Requirements Document using the template that the team believed would be valuable. However, it soon became clear that I was creating a document that the team had no interest in using or reviewing, even though they had asked me to create the document. I would try to schedule reviews of the document, and the team would ignore my requests, or the team would attend and take action items to help me improve the document and then would not follow up. So, on the third unsuccessful attempt at a review, I probed the team deeper on their true need. I wanted to know what they really wanted me to do because a "Product Requirements Document" didn't seem to be the answer. At that time, I found out that what the team really wanted was to show their maturity as a project to certain managers more typically involved in closed-source projects. In closed-source projects, having a "Product Requirements Document" was essential to success.

However, fortunately for me, the managers making this request also understood the difference between a name and a need. On further investigation, I found that a "Product Requirements Document" was not what they required. They wanted to *ensure that the open source project had a solid process in place for*

collecting, prioritizing, implementing, and testing requirements. The further good news was that the team had quite a bit of this process in place already. My job became a matter of finding a way to ensure this process was clearly communicated to all members of the team and their stakeholders. We held a meeting to discuss the true needs, and, from that point on, my job became one of publicizing and clarifying the existing web-based process in place for collecting, prioritizing, implementing, and testing requirements. Once the managers that were part of the project understood the well-defined method for requirements management, the requests for a "Product Requirements Document" went away.

Focusing on the true need allowed us to make progress in this instance. I have found this technique effective in many situations in volunteer armies. For example:

1) Instead of asking the team where their schedule and Gantt chart is, I like to ask "How do we know what we will deliver and by when?" A milestone list on a PPT or web page may be all that is needed to answer that question.
2) Instead of asking the team where our formal "Release Criteria" are documented, I like to ask "How do we know when we will be done?" This is especially effective in volunteer armies. It's empowering to change the answer from "When we are all burnt out." to "When we have accomplished these specific items." Knowing what you are responsible for up front can actually reduce burn out. It also can help change the other answer seen in volunteer armies, especially standards organizations, of "When every single concern with our work product is completely gone." to something more realistic and satisfying to all members.
3) Instead of asking the team who will do QA, I like to ask "How do we know that what we are delivering is good enough?" A volunteer army frequently can't afford an official QA team, but they typically can find the time to ensure they have enough volunteers to provide a reasonable degree of confidence that their end work product is of the level of quality the group desires.

8. Expect respect from the team

In addition to respecting the volunteer army (see section 2), it is also important as a volunteer to expect (and receive) respect from the volunteer army you are working with. Because volunteer groups don't have as many formal rules governing behavior and roles as corporations, it is frequently easier for one person or group of people to dominate the conversation or the agenda. In some instances, this may be welcome. For example, in a standards organization or open source project, the volunteers may all agree on the goal but may not all have the time to dedicate towards accomplishing the goal. They may welcome a few individuals or companies that step up to assist the group in making progress. However, for the situations where the dominance of a few is not welcome, it is important to remember that you as a volunteer should be getting the same respect you give your peers. In these situations, the following techniques may be effective:

1) Jump in. As noted in section 4, sometimes jumping in is also a way to share the workload with others that may be taking on more of the work or expressing the majority of the opinions. Occasionally, this is all that is needed for the person or persons that are taking more of the limelight to step aside and let others in.
2) Confront the issue. Unlike in business situations where there are clear escalation paths and structures for reporting and confronting issues, in volunteer armies you may need to confront the issue with the offending person(s) yourself or together with a few like-minded allies.
3) Tolerate the situation. As noted above, in some cases, a volunteer army may be delivering a quality product even though a few individuals are running the show. If the goals you came to the volunteer army to achieve are being met in general, you may find it acceptable to tolerate the situation because you are achieving the results you wanted to achieve.
4) Leave the volunteer army. People come and go from volunteer armies all the time, and it is easier to leave a position or group where your livelihood and families' financial security does not depend on it. If you find that the situation is such that you cannot simply tolerate it and you are unable to change things by addressing the situation or asserting yourself more, you may find that it is time for you to move on. There are many volunteer armies out there, quite a few of which have similar missions and visions, so it may be time to find a new avenue to express your values and talents.

9. In Summary

Achieving quality in volunteer armies requires patience, persistence, and dedication. However, it is possible to motivate and influence volunteer armies to deliver high quality results. Perhaps the most important lesson to remember is to respect your volunteer army and expect respect yourself and then all other approaches will fall into place. When you expect respect, you are willing to jump in to accomplish goals, and when you respect the team you are excited and eager to solicit their opinions via straw man proposals and lists of tasks needing volunteers. You are also willing to work within the system before suggesting change, and, when you do suggest change, you focus on the true need, not just the terminology or processes you have used before in other positions or organizations. When you do all of these things, you can begin to feel the satisfaction and accomplishment that comes from delivering quality in a volunteer army.

References

[i] Eric S. Raymond. "The Cathedral and the Bazaar." http://www.catb.org/~esr/writings/homesteading/cathedral-bazaar/ar01s11.html
[ii] Wikipedia. "Stone Soup" entry. http://en.wikipedia.org/wiki/Stone_soup.

Sabotaging Quality

Sacrificing Long-Term for Short-Term Goals

Joy Shafer
joyous1@live.com

Abstract

From the very beginning of the project your team has had discussions about quality. You have unanimously agreed it's a top priority. Everyone wants to deliver a product of which they can be proud. Six months later, you find yourself neck-deep in bugs and fifty-percent behind schedule. The project team decides to defer fixing half of the bugs to a future release. What happened to making quality a priority?

One of your partners is discontinuing support for a product version on which your online service is dependent. You have known this was coming for years; you are actually four releases behind your partner's current version. The upgrade of your online service has been put off repeatedly. Now the team is scrambling to get the needed changes done before your service is brought down by the drop in support. You are implementing the minimum number of features required to support a newer version. In fact, you're not even moving to the most current version—it was deemed too difficult and time-consuming to tackle at this point. You are still going to be a release behind. Are you ever going to catch up? Is minimal implementation always going to be the norm? Where is your focus on quality?

Do these scenarios sound familiar? Why is it often so difficult to efficiently deliver a high-quality product? What circumstances sabotage our best intentions for quality? And, more importantly, how can we deliver quality products in spite of these saboteurs?

One of the most common and insidious culprits is the habit of sacrificing long-term goals for short-term goals. This can lead to myriad, long standing issues on a project. It is also one of the most difficult problems to eradicate. There are other saboteurs: competing priorities, resource deprivation, dysfunctional team dynamics, and misplaced reward systems, to name a few. In this paper I'll focus on the quality saboteur of sacrificing long-term goals for short-term goals.

I will discuss the benefits that can be gained when you make the right long-term investments and the types of problems you'll see if your team is solely pursuing short-term goals. I will show you practical strategies for slaying this saboteur or at least mitigating its effects.

Biography

Joy Shafer is currently a Consulting Test Lead at Quardev on assignment at Washington Dental Service. She has been a software test professional for almost twenty years and has managed testing and testers at diverse companies, including Microsoft, NetManage and STLabs. She has also consulted and provided training in the area of software testing methodology for many years. Joy is an active participant in community QA groups. She holds an MBA in International Business from Stern Graduate School of Business (NYU). For fun she participates in King County Search and Rescue efforts and writes Sci-Fi and Fantasy.

1. Introduction

After twenty years as a tester or a test manager, I've seen team after team struggle with the complex problem of delivering high-quality software on time. I've been on teams that have excelled. I've also been on teams that have failed miserably. I've learned a lot from both experiences. There are reasons behind the failures. Sometimes they are even reasonable reasons.

In theory, fixing problems is a straightforward process: (1) understand the problem, (2) brainstorm a solution, (3) implement the solution, and (4) examine your results. If the results are not as desired, go back to step one. Otherwise, celebrate your success and look for the next problem to tackle.

In practice, fixing problems can be very challenging. What seems like the obvious reason for a problem may not be the reason at all. There are usually layers of issues, not the least of which are roadblocks thrown up by people. Individuals often have their own reasons for fighting change, which usually have nothing to do with purposefully sabotaging success.

In this paper I'll examine these saboteurs—not the people, but the reasons they may have for undermining quality—and discuss strategies for mitigating them.

2. Sacrificing Long-Term for Short-Term Goals

Sacrificing long-term goals for short-term goals is probably the most common problem and also the most difficult to eradicate. There are usually compelling arguments for investing in short-term goals. Some of them are quite justifiable. However, if your team consistently sacrifices long-term goals for short-term goals, you'll find yourselves unable to deliver efficiently even on those short-term goals.

The types of things that tend to be sacrificed are:

1. Investment in test, particularly test automation
2. Software quality and maintainability
3. Investment in infrastructure
4. Employee morale and vitality

2.1 Investment in Test Automation

It has been my experience that investment in test, particularly in test automation, pays off handsomely in the long run. In order to have an efficiently running development engine you'll need to have the following:

1. Automated Build Verification Tests (BVTs)
2. Continuous Integration
3. Unit testing
4. Automated regression tests
5. The ability to emulate external components

2.1.1 Automated Build Verification Tests (BVTs)

Build verification tests are those mainstream tests that you run every time you get a new code drop in order to verify that your code is functional and stable enough to test.

It is imperative that you have a robust set of BVTs. By robust, I don't mean extensive. You can start with a minimum suite and add on as desired. What I mean by robust is reliably functional. Not brittle. Not dependent on anything other than the code to make them pass. The test cases should be definitive: if a

test fails it's because of a bug in the code, not because of a timing issue, a setup issue, a data issue, or some other non-code problem.

One of the teams I joined had BVTs that ran daily. Great! Except the pass rate for those BVTs was less than sixty percent—every day—for years! What was going on? Did no one care?

When I started asking questions, I was told, "A lot of those failures are because our service is dependent on other teams' services. They don't keep their integration environment stable, so when they are down, those tests will fail."

Other tests were failing because the person who used to maintain them had left the team and hadn't been replaced yet. (It's no surprise that this team was having trouble hiring and retaining good people.) Still other tests were failing due to hardware issues that no one had time to troubleshoot.

The clean-up process for this mess was excruciatingly slow, partly because, as you can imagine, failing BVTs was the tip of the iceberg for the deep-seated problems on this team. You'll hear a lot about them here, because they make a very good example of how a team can sabotage quality.

One of the things we did to mitigate these problems was to build emulators[1] for as many of the external dependencies as we could. The merits of building emulators warrant their own discussion later in this paper (section 2.1.5, Emulate External Components). We also made sure each BVT had a tester responsible for maintaining it, and we cleaned up the test lab environment—also discussed later (section 2.3.1, Maintaining your Physical Environment).

Creating BVTs that are not definitive is a common problem. I worked on a team where the data in their automation, including BVTs, was real customer data that was refreshed in the test environment on a regular basis. The problem was that testers could not depend on a particular piece of data being in the same state every time. This caused numerous failures in the automation with each data refresh.

When I pointed out that the automation should create its own data and then clean up after itself when it was done, I was told that that would be too difficult and time-consuming. The business rules were too complex. The team would never be able to recreate the exact conditions they needed.

I don't believe that. It would take an upfront investment in time, which considering how much time was spent by testers working around this issue with every database refresh, was easily justifiable. It would also take expert knowledge of the business rules, which I suspect may have been missing from the test team, but was certainly available somewhere in the company. That knowledge could and should be captured and transferred to the test team. These problems are solvable, once you get to the root of them and figure out how to get around the personalities involved.

2.1.2 Continuous Integration

I cannot overstress the importance of automated BVTs. Without automated BVTs, there can be no continuous integration.[2] Continuous integration is the process of checking in code and creating a new build on a continuous or near-continuous basis. It is a cornerstone of efficient software development. For continuous integration to work well, once the build is created, it should be automatically deployed to a test

---

[1] Two of the dependencies didn't get emulators, one of them because the service was extremely complex and regularly updated, so an emulator would have been difficult to build and maintain; that service also had a solid track record of keeping their integration environment stable, making an emulator less critical. The other service did not get an emulator because it was being deprecated.

[2] Martin Fowler, "Continuous Integration," *MartinFowler.com,* May 1, 2005, http://martinfowler.com/articles/continuousIntegration.html.

environment where the automated BVTs will kick-off. The developer will get very timely feedback on whether she has broken the build.

If your team does not integrate often—I recommend at least daily—you run the risk of building bad code on top of bad code. When you do finally integrate it will be more difficult to determine the causes of failures. Your development timeline will be much less predictable (read: you're going to slip, probably multiple times).

I worked with a team who only integrated their code once every few months. I was astounded that they ever shipped anything. By contrast I worked on a globally distributed team that integrated twice daily with an automated build/deployment/test system. It was the most efficient team on which I've ever worked.

2.1.3 Unit Testing

Unit testing is another activity that is often sacrificed due to 'not enough time.' In reality, you don't have enough time to forgo unit testing. Unit testing provides more than enough benefits to make it a justifiable activity.[3] Among these benefits are:

1. Cleaner code: Not only is the code less buggy, but it is often simpler and more elegant. The process of unit testing gives the developers a different perspective on their code. They may well come up with a better design in order to accommodate the required unit testing.
2. Tools: To facilitate their unit tests, developer will create test hooks, mock objects, and other devices very useful to the test team. Test and development should work closely together to share resources and knowledge.
3. Team Building: The developers acquire a better appreciation of the challenges testers face.

How do you start unit testing if your team has never done it before? The first thing you'll need, as is the case with many of these recommendations, is management support. Your managers need to communicate to the development team that unit testing is required—not optional. To facilitate the on-boarding process, you'll want to implement an appropriate measurement and reward system.

One of the teams I was on successfully added unit testing to its development efforts by insisting that code coverage be run on all code before it was checked in. The unit tests had to cover at least 70% of the code. This process was implemented for all new code as well as for every bug fix in the legacy code.

This worked very well. There were some holdouts on the development team, but statistics were made visible and soon there was a small competition among the developers for who could hit the highest coverage rate on new code check-ins.

Within a few months of implementing this process, our code had gone from zero unit tests to almost 16% of the code being covered by unit tests. Added to the other best practices this team was following, our development timeline became extremely predictable. We were held up as the 'poster child' for development efficiency.

2.1.4 Automated Regression Tests

Automated regression testing is a key component of an efficient and successful software development process. Without automated regression testing, teams of manual testers will need to comb through the software with every release, laboriously running the same tests they've run countless times before. In this case, it becomes easy to cut corners. You haven't seen a particular test case fail ever. Your manager is

---

[3] J. Timothy King, "Twelve Benefits of Writing Unit Tests First," *J. Timothy King's Blog,* July 11, 2006, http://blog.jtimothyking.com/2006/07/11/twelve-benefits-of-writing-unit-tests-first.

breathing down your neck to sign-off. You skip the test this time, marking it 'pass'. Oops! This time it would have failed.

Automation doesn't lie. Well, sometimes it gives a false positive, but well-written automation will not give a false negative. Bugs that slip by the automation tend to be the ones for which a test case was never created.

Automation is also many times faster than manual testing—once it's written. It becomes a no-brainer to run whatever automation you have on a regular basis, perhaps at night, to ensure that your continuous integration stays stable.

Sometimes it's difficult to justify the expense of automation, especially if your team is starting from scratch. However, if you have an efficient test development team, equipped with proper knowledge and tools, you'll find that the return on investment (ROI) for writing automated tests usually exceeds the ROI for manual tests, certainly within a few years and sometimes within six months.[4] Other benefits include being able to run tests that otherwise could not be run (better coverage), a more interesting and rewarding experience for the testers, and an increase in the agility of the development effort—test automation supports other best practices, such as continuous integration and code coverage measurement.

2.1.5 Emulate External Components

The concept of testing all the various parts of the system in isolation has been around for almost as long as software development. It used to be done with drivers and stubs. In vogue now are mock objects,[5] although, if done on a component level they are sometimes called emulators, simulators, or sinks.

It may seem like a daunting undertaking to build an entire component that mimics the functionality of one of your dependencies. However, you typically don't need to build in all the functionality, or even build much at all. You just need to create a mechanism that takes input and gives appropriate output. There is no need to have it actually pull real data on the backend or anything close to the complex functionalities that the real software or service actually accomplishes.

If you're working for a large company, I recommend asking around. Some other team may well have already built the emulator you need. Perhaps with a few modifications it will work for your purposes as well.

An added benefit of emulators is they are extremely handy—often essential—for performance testing. It is not uncommon for emulators to be built specifically for this purpose, typically near the end of the project when the team finally gets around to performance testing. However, you'll find if you build the emulators up front, they are useful in every phase of the project, including unit testing and BVTs.

2.2 Software Quality and Maintainability

2.2.1 Maintaining Software Quality

Putting off bug fixes to the end of the project is another tactic that falls under the category of sacrificing long-term for short-term.

---

[4] Douglas Hoffman, "Cost Benefit Analysis of Test Automation," paper presented at STAR99, 1999, http://www.softwarequalitymethods.com/html/papers.html#CostBenefit.

[5] Steve Freeman and Nat Pryce, *Growing Object-Oriented Software Guided by Tests*. (Boston: Pearson Education, 2010).

When you are running up against your deadline and there is still a mountain of open bugs in your software, you will likely either slip or ship with bugs. It is common for teams to postpone bugs that normally would have been fixed because they don't want to miss their deadline. Keep in mind that for every known bug, there are also unknown bugs. If you ship with lots of known bugs, you run a higher risk that end-users will find even more bugs, some of which could be very costly.

When a team is close to shipping and behind schedule, it is very difficult to justify bug fixes, especially since they may destabilize a mostly-working software system. The solution is to fix bugs as you find them.[6]

There are a number of advantages to doing this:

1. You won't carry a backlog of bugs from release to release, causing the next project to start with a "bug debt." If your team lets the backlog get too high by pushing bug fixes to the next release, you'll eventually get to the point where the software is almost unmaintainable and the backlog is so large that it's daunting to even understand the extent of the work ahead.
2. Your development cycle will become more predictable. Some bugs are notoriously difficult to fix. If you put these off to the end and then discover a doozy—you're slipping.
3. Bug fixes may introduce new failures. If bugs are not fixed until the end of the project, then there is less time for the fallout bugs to be found. You will likely ship with some undiscovered, possibly serious, bugs.

Sometimes, with a particularly buggy area of the software, what makes the most sense is to refactor it instead of patching it up with individual bug fixes. Code refactoring means restructuring or rewriting an existing body of code without modifying its functionality. It's done to increase the maintainability of the software by making it more readable, extensible, modular and/or less complex.

Really buggy areas usually have high-cyclomatic complexity.[7] Cyclomatic complexity is a software metric that measures the number of linearly independent paths through a program's source code; some code coverage tools include the ability to measure cyclomatic complexity. Software with high cyclomatic complexity is typically also difficult to maintain, arduous to test well, and very brittle—poke it here, it bulges there.

Whether refactoring an area of the software is justifiable is a decision that needs to take into account how much time is spent dealing with the backlash of the bad software. Does it directly affect customers, and therefore customer support, marketing, the live site support team, and so on? Do the developers avoid making changes in that area because of the time and risk involved? Are lots of resources being spent because of the buggy code? Is it in a component that's going to be required for a long time yet to come? Is the technology obsolete? Will it not be supported with future versions of the operating system or database? If the answers are "yes," it may be worthwhile to refactor.

Even if refactoring doesn't make sense, keeping bug counts low on your software does.

For teams that already have a high backlog, I recommend doing a "quality" release—or maybe even several quality releases, depending on the size of your backlog. The focus of a quality release is on fixing bugs and/or refactoring areas of your software. New features should not be permitted into the quality release.

---

[6] James Shore, *The Art of Agile Development*, (Sebastopol: O'Reilly, 2007), http://jamesshore.com/Agile-Book/no_bugs.html.

[7] Julias Shaw, "Cyclomatic Complexity—What Is It and Why Should You Care?" *Java Metrics,* April 15, 2010, http://java-metrics.com/cyclomatic-complexity/cyclomatic-complexity-what-is-it-and-why-should-you-care.

For teams that are mostly keeping up, or teams that are using an agile approach, one technique that works well for keeping bug counts low is a 'bug jail'. Any developer that has more than three or five or eight – you decide—bugs open against them isn't allowed to develop new code until they get their bug counts below the threshold. You might opt to move the threshold to a lower number the closer you get to release. Bugs should be fixed as they're found, not put off to the end of the project.

2.2.2 Software Maintainability

Maintainability is generally defined as the ability to make modifications to the software after delivery. I'd like to expand this definition a little to include not only software modifications, but also deployment—something you'll have to do if you make changes. How easily your software can be modified and deployed makes a huge difference in how much time you'll spend on maintenance after your software or service is on the market.

When building a version 1.0 software or service, it's easy to overlook maintainability as a goal for the product. Especially since new product development tends to be very time-sensitive—those products first on the market gain the largest market share—there will be lots of pressure from management to ship quickly. But if you keep maintenance in mind during the design phase of your product, you will save your team lots of pain and drudgery down the road.

As early as possible in the life of your product, start thinking about investing in the maintainability of your software or service. If not, you could find yourself in a situation similar to one of the teams I was on.

This team developed a service that supported cell-phone technology and, as you can imagine, our user-base grew exponentially. Unfortunately, the team had not spent any resources on maintainability for the service. As a result, when I joined the team, they were spending about eighty percent of their time trying to maintain the software and only about twenty-percent of their time was available to develop new features and meet partner requests.

Here is a litany of the issues facing this team caused by not investing in maintainability:

1. The live site was unstable

   This was the most insidious of the issues facing this team. Whenever there was a live site issue, which was almost daily, it was 'all hands on deck'. Whatever resources were required were yanked off of the project they were working on and told to fix the live site.

   The service had millions of users. Outages caught the eye of senior management and caused screams from the poor customer service technicians who had to deal with the irate customers. Not to mention the affect it had on our bottom line: fully forty percent of the people who tried the service abandoned it within the first few months. Once we stabilized the live site, this figure plummeted to eleven percent.

2. Deployment was a nightmare

   In addition to resources being poured in to maintaining the brittle live site, every time a fix for a live site issue was developed, it needed to be deployed. Not just deployed once to the live site, but also deployed to several different test environments for various purposes: BVT testing, functionality testing, integration testing, performance testing, and staging environment testing.

   Unfortunately, the deployment engine had been designed with a single deployment in mind. Every time a change was needed to deployment, not a rare occurrence, the change would need to be manually entered into a huge array of 'cluster' scripts. Of course, the changes never made it cleanly into all of the environments and countless hours were spent with each and every deployment troubleshooting what went wrong.

Because the deployment system was so prone to problems, a lengthy process had developed around deployments to the live site. For each release a long detailed document was written to support the deployment. The deployment was tested numerous times in various environments, and the document was edited, updated, and approved. Even this arduous process didn't prevent deployment issues in production.

By the time I joined the team, the deployment mechanism was so huge and complicated, that to overhaul it into an efficient system would have been a large project on its own. This deployment project, I am sure, would have paid for itself within a few releases, but it was not supported by management. Thus, a huge amount of resources continued to be sucked into the maintenance and testing of the deployment engine.

3. The physical infrastructure was unreliable

   When I started on this team, fully fifty-percent of the machines in use in the test lab were past warranty. Upon investigation, I discovered some of the network routers were more than ten years old.

   Both the physical infrastructure and the software service on this team were poorly maintained. Every time there was an issue, the IT department called in the devs to look at it. Logging had not been added to the code for even the most basic troubleshooting. The devs would very quickly throw up their hands and push the problem back to the IT department. This friction added more stress to an already unhappy situation.

   A tremendous amount of resources were wasted on troubleshooting issues that would never have occurred in the first place in a well-run, properly-maintained test lab. Although not a software maintainability issue, the poorly maintained lab added significantly to the time the team spent on non-productive activities. This topic is covered further in section 2.3.1, Maintaining your Physical Environment.

Dealing with the above three issues probably took up about seventy percent of the team's available time. With proper investment in maintenance and infrastructure, teams should be efficient to the point where less than twenty percent of a team's time is taken up in maintenance activities.

2.3 Investment in Infrastructure

Surprisingly little has been written about investing in infrastructure. It has been my experience that investment in infrastructure almost always pays for itself, sometimes fairly quickly. By infrastructure, I mean the systems that support your core activity. The core activity is software development. In support of this are the physical infrastructure: servers, networks and software; logging and error handling infrastructures; the deployment system; development methodology and processes; and metrics. I'm going to discuss three of them here:

1. Physical Infrastructure
2. Logging
3. Metrics

2.3.1 Maintaining your Physical Environment

Maintaining your hardware and network system should not even be something there is a serious discussion about. Just do it!

Often when hardware goes bad, it does so spectacularly, with system beeps, blue screens, black screens or even smoke. Sometimes it simply won't power-up any more. These problems are generally easy to identify and remediate. However, sometimes when hardware goes bad, it does so discretely. It fails

intermittently. Or it doesn't fail completely, it just slows way down. Or it starts sending cryptic errors to other systems which cause them to fail but don't necessarily lead the troubleshooter back to the root cause of the problem. A ton of time can be wasted trying to troubleshoot hardware that should have already been scrapped.

I took over the test lab on the above referenced team shortly after I started there. One of the things I discovered very quickly was, in addition to the ancient hardware, patches were not applied on a regular basis. Many of the machines in the lab had not been patched in years. I was astounded that the team had been able to get away with this.

I was told the reason for the deplorable state was because the lab was always too busy. The IT people were never allowed the time that was needed to bring the lab down to get everything updated. With this team being in perpetual crisis mode, I could understand that mentality on some level. However, we needed to turn it around.

I coordinated with the appropriate people, over-communicated the changes we were about to make, and then implemented a process whereby once a month, the lab was unavailable from 6am to 10am for patching. This change went smoothly, and as it turned out, happened in the nick of time.

One of the network engineers, someone not on our team but on a different corporate IT team, accidentally created an open channel from our test lab to the Internet. This was discovered fairly quickly by corporate security and our lab was shut down. A security team came in and did an audit of our systems.

They were impressed that we were completely up to date with our patching, and as a result, released us to normal operations within 24 hours. If they had done the same audit two months earlier, there likely would have been some very negative consequences.

2.3.2 Logging

Some of the reasons to build logging into your software or service include providing the ability to troubleshoot issues, supplying a means to prove to partners that you're meeting your Service Level Agreement (SLA) requirements, and determining customer usage patterns.

Without appropriate logging, it may be almost impossible to troubleshoot an issue. Often logging is added by a team simply because they've run into a problem and cannot fix it until they understand it, and can't understand it without more information.

Logging can be instrumental in alerting you to brewing problems with enough advance notice to proactively prevent a crisis. A common example of this is impending performance issues. Logging and analyzing the appropriate statistics can let you know that, if the customer base continues to grow at the present rate, you have about six months before your live site will start to melt down.

Logging and analyzing customer usage patterns can help you design software to better meet their needs, market the software more effectively, and design better real-world test cases.

Ideally logging infrastructure is designed and built along with the initial development of the software. Logging itself can be added incrementally, but the structure and policies surrounding it should be in place from the beginning. Logging is a great example of a long-term investment that can provide tremendous benefit. Figure out how to squeeze it in.

2.3.3 Metrics

The main goal of collecting metrics is to understand the health of your system and processes so you know where to invest in improvements. A secondary goal of metrics, and one that many people overlook or underestimate, is to drive desirable behavior.

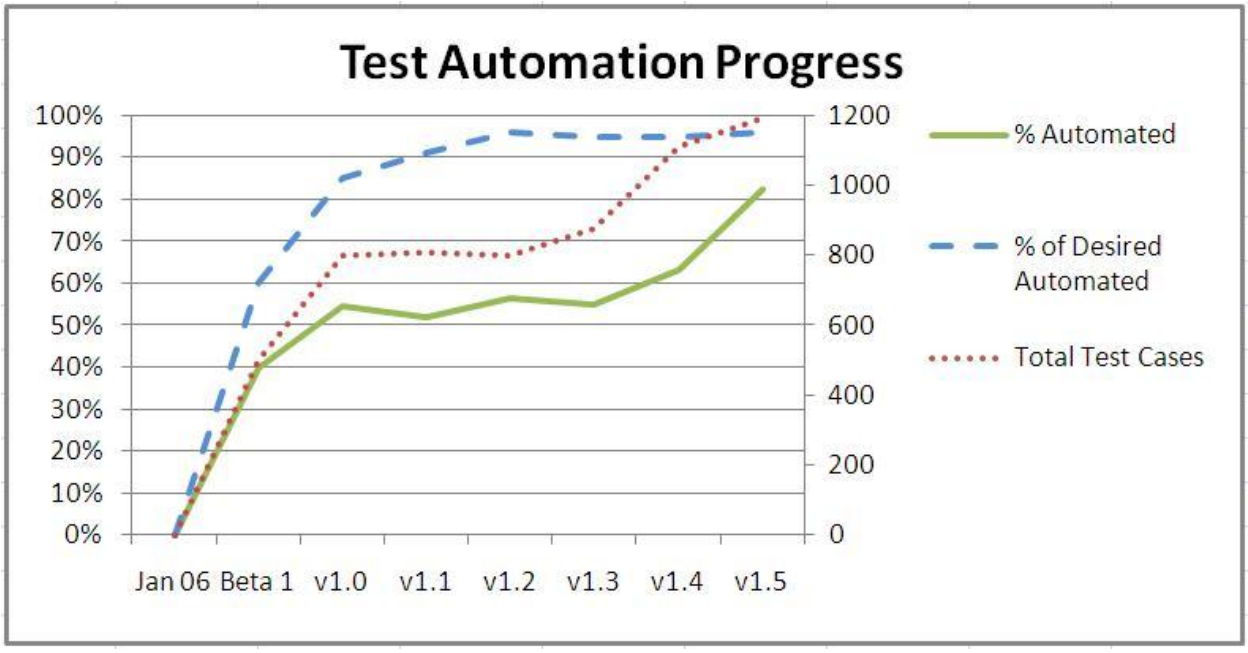
I'm not going to talk about which metrics you should track here, because that's not the focus of this paper. What I do want to mention is that some metrics take time and effort to collect. It is often difficult to make the argument to invest in metrics because they tend to be a long-term payoff activity. However, the teams I work with that have well conceived, well analyzed, and appropriately communicated metrics are also the teams who deliver the highest quality software on time.

You have to be careful with metrics, because you may end up driving undesirable behavior. For example, on one of the teams I was on, to help me determine which test metrics to track, I sent an inquiry to one of the test lead aliases asking which metrics others tracked, and specifically, how best to track test automation.

I got only a few positive responses. Mostly what I got was a barrage of advice from indignant test leads saying I should not even try to track such metrics because they would be used against me. I definitely thought it was worthwhile to track some type of metrics, but armed with all these dire warnings, I decided not to track the percentage of test cases that were automated, but rather to track 'of the cases that we want to automate, what percent is automated'?

At the time most of my test team was in India, and we were developing a couple of different brand new services. We went from zero to 95% automation, based on my statistics, within eight months. Thereafter we stayed about the same. I was pretty happy with the results until I looked at the total percent of test cases automated—it was running under 60%. This didn't feel right to me; I was sure that we should be automating a higher percentage of overall test cases.

I changed the reporting requirements to include the 'number of test cases' and 'percent of total test cases' as well as the previously collected 'percent of what we desire to automate.' Immediately thereafter both our total test cases and our automation percentage started to climb. Fairly quickly the team had automated 82% of the total test cases. The reporting change took place starting with v1.4 as shown below.



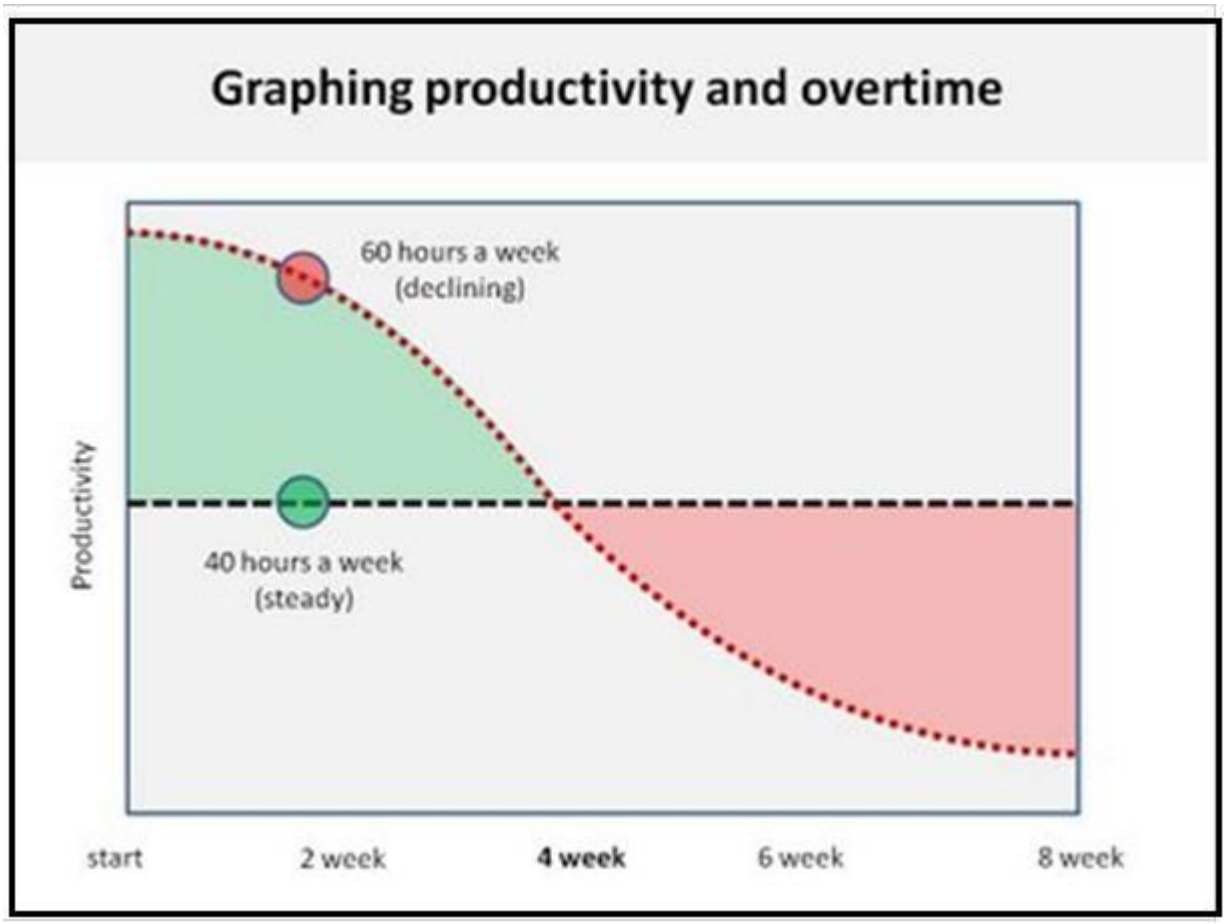
2.4 Employee Morale and Vitality

You're behind schedule, the ship date is looming, and it's too late to bring more people on board to help out. The obvious solution is to get the people you do have to work harder, smarter, and longer hours. On paper this looks like a viable option and, indeed, it usually works in the very short-term if you have a

healthy team. Give your team a pep talk, get everyone to come in on the weekend, and viola, the work is done a few days earlier than it otherwise would have been done.

The problem is that working long hours can become a habit very quickly—a habit that is extremely destructive. For some teams, sixty-plus hour weeks are the norm. These teams tend to have myriad problems, not the least of which is productivity, in spite of the long hours. Numerous studies have shown that working fewer hours may increase productivity[8] and working longer hours actually decreases productivity.[9] I strongly suspect that an analysis of productivity by hours worked will show the classic bell curve.

Danc has a very interesting slide show on this topic on his blog where he depicts the following chart. He claims that the productivity loss during the recovery period, even for a relatively short time spent in crunch mode, is usually greater than the productivity gain during crunch mode. He also states that the teams who were working longer hours *reported feeling more productive* than those working a normal schedule, although when measured objectively they were less productive.



[8] Robert LaJeunesse, "Toward an Efficiency Week—Correlation between Workweek and Higher Productivity," *The CBS Interactive Business Network,* (Jan-Feb, 1999), http://findarticles.com/p/articles/mi_m1093/is_1_42/ai_53697784/pg_2/.
[9] Danc, "Rules of Productivity Presentation," *LostGarden*, September 28, 2008, http://www.lostgarden.com/2008/09/rules-of-productivity-presentation.html.

In addition to defeating the purpose of working all those hours in the first place, there are other problems your team is likely to face if you don't keep working hours reasonable. These include high turnover, burn-out, demoralization, and lack of trust in management.

Most of you are probably familiar with the classic project resource triangle: time, money and quality. If you want to decrease the time to develop your software, you will need to increase the money (resources) and/or decrease the quality (fewer features, less testing). The part of this equation that is not always mentioned is that, at the end of the project, resources should not be depleted. You should be able to go right into the next project without recovery time. So, if you're in manufacturing, you should not be completely out of parts. In software development, your people resources should not all need to take a vacation.

I was on a project where, after literally working 60-80 hour weeks for over a year, five out of seven testers quit the team as soon as we shipped. (This became my first management job—I was the only one left except for the new guy.)

3. How to Get People to Follow Best Practices

I've gone on and on about best practices, and now you're thinking. "So, yeah, we know they work. That's why they're called 'best practices'. We're familiar with those best practices, on a conceptual level at least, because we're professionals and we've taken the time to understand what we should be doing to get it right. What we want to know is *how* to actually get to the point where we're doing those best practices instead of just talking about them. Something is always in the way."

What is usually in the way is people: people who don't know better; people who have competing priorities; people who have more power and influence than you. How do you get around that?

You need to understand the motivations people have for doing what they do and wanting what they want. Only then will you be able to effectively persuade them to change their minds.

3.1 Individual Goals may not Sync with Team Goals

Every single person on your team has their own personal goals, either stated or unstated. Sometimes these goals are in line with team and corporate goals, sometimes they are not. Sometimes hidden agendas of team members may wreak real havoc on your project.

A case in point is the project that I've used as an example throughout this paper (I knew going through that hell would give me some mileage somehow!). In this case, it took me a long time to figure out what was really going on.

Once I understood the magnitude of the problems facing this team and we started making progress toward fixing them, I found myself undermined at every turn—sometimes very subtly, sometimes more bluntly.

It turns out that the release manager thrived on crisis management. It was his moment to shine. He loved being pulled out of bed on Christmas morning because the live site was melting down. He had gotten kudos his entire career because he was always there when he was needed, willing to work tirelessly sixty-plus hours a week, sacrificing precious time with family and friends to put the corporate needs first.

But was he really putting corporate needs first? When we started to see change and, after the first few releases, the live site stabilized, he panicked. He was a mediocre manager in the best of times. He needed the adrenalin. He needed to be desperately needed. He needed a crisis.

And he was entrenched. He had been on the team a long time. He had lots of friends and the support of his manager. In spite of the fact that the service was in horrible shape, he had always been told he was doing a good job.

Your first defense in a situation like this is to be aware of it. Any time you start making changes to the status quo, you are going to make some people unhappy for whatever personal reasons they may have. They will fight you.

Tread carefully, but figure out who you need to convince to make sure needed changes happen. Don't neglect to consider that even positive changes may be seen in a negative light by some team members.

3.2 Reward Teamwork

Unless the project is tiny, software development is very much a team effort. The best way, ultimately, to consistently deliver high quality software on time, is to have a reward structure in place that rewards teams for behavior that leads to successful releases.

This is common sense, however, I've seen very few companies who reward team effort rather than individual effort. Perhaps the fear is that if you reward the team, the individuals on the team won't work as hard because their personal effort is not being recognized. On most of the highly functioning teams on which I've worked, this would not have been the case. People will work even harder because of the team than they will on their own. The team is motivating; everyone works hard so that they won't let their teammates down.

If you can't get official corporate policy to support rewarding team effort, try to get your management to at least support team effort with recognition at meetings, extra days off, a team party or special function or something else the team desires.

Luckily, working on a successful team is a rewarding experience in itself. My best work experiences are those in which I am an integral part of a tight, high-performance team.

3.3 Tactics

There is no one way to accomplish your goals that will work for every situation, but I've found specific tactics very helpful:

1. Know best practices

   If you don't already know what the best practices are in software development, take some time to learn about them.[10] Best practices work.

2. Be polite but persistent

   Gently encourage team members toward best practices. You'll find if you're too heavy-handed you'll usually get beaten down. Take every opportunity to educate the team and management on best practices and why they should be followed. If you're persistent, eventually people will come around.

[10] Joel Spolsky, "The Joel Test: 12 Steps to Better Code," *Joel on Software,* August 9, 2000, http://www.joelonsoftware.com/articles/fog0000000043.html.

3. Find your allies

   There are usually others who feel the same way you do. Enlist their help in fixing the problems on your team. They will often be delighted to find a like-minded person to help them with the frustrating problems with which they've been dealing, and if they've been on the team for a while, they may have some excellent insight into the reasons things have not yet been fixed.

4. Understand the real problem

   It's fairly easy to look at a team and say, "You're not doing this and that. Start doing this and that and your problems will be solved." There is a usually a reason they are not doing this and that already. You need to discover that reason, otherwise you will not be able to get around it.

5. Measure yourself

   I'm not just talking about bug metrics here. Measure your team on how well it does in meeting best practices. Take inventory. What are your current practices? Compare them to best practices. How far away are you and what steps do you need to take to get there?

6. Set clear goals

   This should be a natural result of measuring yourself. Figure out what is causing the most pain and fix that first. If the first solution you try doesn't work, try something else.

7. Celebrate successes

   Once you start seeing success, reward it. Brag about it at company meetings. Have a party. Give away special prizes. Tell the team how proud you are of their accomplishments. Usually success gets people motivated to go on to even more success.

Most importantly, don't give up!

4. Conclusion

Many things can sabotage quality. The most common is sacrificing long-term goals for short-term goals. Many best-practice activities require long-term investment. If your team is struggling with quality, you need to evaluate your situation and decide which best practices to implement first.

To convince the team to make the needed investments, you'll need to understand their motivations. Only if you know the real reasons behind resistance to change, will you be able to effectively persuade your coworkers, banish the saboteurs, and adopt more effective practices that will lead to higher quality, on-time releases.

Happy development!

References

Bach, James. *Satisfice, Inc.*, last accessed August 14, 2011. http://www.satisfice.com.

Bird, Jim. “Zero Bug Tolerance.” *DevOps Zone,* July 11, 2011. http://agile.dzone.com/news/zero-bug-tolerance-intolerance?mz=38541-devops.

Cohen, Bram. “How to Write Maintainable Code.” *Advogato*, March 15, 2001. http://www.advogato.org/article/258.html.

Danc. “Rules of Productivity Presentation.” *LostGarden*, September 28, 2008. http://www.lostgarden.com/2008/09/rules-of-productivity-presentation.html.

Fowler, Martin. “Continuous Integration.” *MartinFowler.com,* May 1, 2005. http://martinfowler.com/articles/continuousIntegration.html.

Fowler, Martin. “Mocks aren’t Stubs.” *MartinFowler.com,* January 2, 2007. http://martinfowler.com/articles/mocksArentStubs.html.

Freeman, Steve, and Nat Pryce. *Growing Object-Oriented Software Guided by Tests*. Boston: Pearson Education, 2010. http://www.mockobjects.com/.

Hoffman, Douglas. “Cost Benefit Analysis of Test Automation.” Paper presented at STAR99, 1999. http://www.softwarequalitymethods.com/html/papers.html#CostBenefit.

Kaner, Cem, James Bach, and Bret Pettichord, *Lessons Learned in Software Testing*. New York: Wiley, 2002.

King, J. Timothy. “Twelve Benefits of Writing Unit Tests First.” *J. Timothy King’s Blog,* July 11, 2006. http://blog.jtimothyking.com/2006/07/11/twelve-benefits-of-writing-unit-tests-first.

LaJeunesse, Robert. “Toward an Efficiency Week—Correlation between Workweek and Higher Productivity.” *The CBS Interactive Business Network,* Jan-Feb, 1999. http://findarticles.com/p/articles/mi_m1093/is_1_42/ai_53697784/pg_2/.

McConnell, Steve. “Gauging Software Readiness with Defect Tracking.” *Best Practices IEEE Software* 14, no. 3, (May/June 1997). http://www.stevemcconnell.com/ieeesoftware/bp09.htm.

Prasadrao, Saikalyan. “Cycolmatic Code Complexity Analysis for Microsoft .NET Applications.” *The Code Project*, September 21, 2005. http://www.codeproject.com/KB/architecture/Cyclomatic_Complexity.aspx.

Rob. “How Can a Shorter Working Week Increase Productivity?” *Beyond Norms,* January 2, 2011. http://www.beyondnorms.com/index.php/2011/01/how-can-a-shorter-work-day-increase-productivity/.

Saktheee2008, “The Law of Diminishing Marginal Productivity of Labour, *HubPages*, accessed August 14, 2011, http://hubpages.com/hub/THE-LAW-OF-DIMINISHING-MARGINAL-PRODUCTIVITY-OF-LABOUR.

Seshadri, Shyam. “The Advantages of Unit Testing Early.” *Google Testing Blog*, July 15, 2009. http://googletesting.blogspot.com/2009/07/by-shyam-seshadri-nowadays-when-i-talk.html.

Shaw, Julias. “Cyclomatic Complexity—What Is It and Why Should You Care?” *Java Metrics,* April 15, 2010. http://java-metrics.com/cyclomatic-complexity/cyclomatic-complexity-what-is-it-and-why-should-you-care.

Shore, James. *The Art of Agile Development*. Sebastopol: O’Reilly Media, 2007. http://jamesshore.com/Agile-Book/no_bugs.html.

Spolsky, Joel. “The Joel Test: 12 Steps to Better Code.” *Joel on Software,* August 9, 2000. http://www.joelonsoftware.com/articles/fog0000000043.html.

Vaaraniemi, Sam. “The Benefits of Automated Unit Testing.” *The Code Project,* Nov 8 2003. http://www.codeproject.com/KB/architecture/onunittesting.aspx.

Vandegriend, Basil. “The Importance of Maintainable Software.” *Basil Vandegriend: Professional Software Development*, February 1, 2006. http://www.basilv.com/psd/blog/2006/the-importance-of-maintainable-software.

Wikipedia. s.v. “Zarro boogs.” last modified August 6, 2011. http://en.wikipedia.org/wiki/Zarro_boogs.

Increasing Software Quality with Agile Experiences in a Non-Technically-Focused Organization

Aaron B. Hockley
aaron.b.hockley@multco.us

Abstract

Agile development methodologies are well-documented but most of the textbook examples and anecdotes found on the internet provide stories of the use of agile methodologies as part of a technically-focused organization such as a software development contractor, retail software company, or hardware/systems organization with software teams that support the company's technical products.

Multnomah County (Oregon) is not such an organization. The county's business consists of providing services such as health services, jails, taxation, licensing, animal control, and managing individuals on parole and probation. A few small software teams build and support tools to support the county's business, but the organization is decidedly focused on non-technical ventures. In a non-technically-focused organization, the role of the product owner (internal customer) becomes challenging. Software development participation competes with their other (usual) job activities and the product owners are often unfamiliar with software development experiences.

Over the past three years, a variety of agile practices have been introduced at Multnomah County; lessons learned by the county's Public Safety Development Team have resulted in software that better meets the customer's needs. Experience has shown that quality improves when the development team has frequent access to business personnel even though the ideal co-located customer scenario cannot be achieved. Communication with the business product owners is key; the county's software teams tried a variety of agile work tracking and communications systems before finding one that works well for all parties. No tool is perfect. The team concluded that given the challenges of customer time and participation, the tool which provides the best customer experience is probably the best tool.

Experience demonstrated that it wasn't feasible to use a textbook Scrum approach due to customer availability challenges. Project teams and customers eventually settled into a development process that's a hybrid between a Scrum approach and a Kanban-style system. This nimble development cycle is working well for all involved parties.

Better software means that the core business services are better met. Software that best supports the staff providing day-to-day county services is important; a nimble development approach enables applications to be created that best meet the staff and resident needs.

Biography

Aaron Hockley is a QA development analyst with the Public Safety software team at Multnomah County in Portland, Oregon. He is actively involved in both day-to-day software testing (both manual and automated) as well as process improvement for software development in the public safety business areas. Over the past ten years, he has worked for a variety of software and technology companies with a focus on software development and testing.

1 Introduction

When we hear or read stories and textbooks of agile software development, the information is often presented from the idealized position that involves a software development team working for a software company. A product manager usually plays the clearly defined role of product owner; the entire organization understands software development and things move forward with only the agile manifesto (Beck et al. 2001) and market forces as guidance. In such an environment it's easy to look at the bottom line and market share analysis to measure a company's position.

The Multnomah County government, based in Portland Oregon, is not such an organization.

1.1 About Multnomah County's Software Development Situation

The business goals of the county are to provide services for its citizens. Instead of an organization that focuses on delivering technology, the county is ultimately tasked with managing jails, public health, libraries, prosecuting criminals, and other miscellaneous services such as animal control. While software and technology solutions are used by county employees (and citizens) in the delivery of these services, the county's business is government as opposed to technology.

The work of the county is performed by well over 4,500 employees; only 40 of those (less than 1%) are software developers. With the development staff representing such a tiny portion of the overall workforce, effective software development is barely a blip on the overall radar of decision makers. A change in library operating hours will consume far more public attention and staff time than whether the county's software engineers are using Scrum, XP, waterfall, RUP, or any other development methodology or toolset.

Development personnel are organized into four teams based on business area. The Public Safety Development Team is the largest group and provides support for the Sheriff's Office (jail management) as well as the Department of Community Justice (parole, probation, and juvenile offenders). While a few off-the-shelf applications are used for some areas of the business, most of the software used to manage offender caseloads, adult and juvenile detention (jail), and treatment and counseling programs are custom application development performed by the county staff on the Public Safety Development Team. Over the past three years, the Public Safety group has become increasingly agile and has found a nimble solution for better software to help the county meet its public service goals.

1.2 Past Challenges with Waterfall Development

In 2008, development staff and management broached the idea of a change to a more agile development process. Existing applications were built using a traditional waterfall-style method. The last large project that was run in such a manner spanned over two years with heavy up-front requirements and design followed by a period of development. After development, customers were presented with the "finished" result. Because over a year had elapsed from the time of initial requirements gathering, business processes had changed. Decisions made by the software development team during construction weren't always correct due to differing interpretations of the requirements documents. After presenting the software to the business group, several more months of work were involved in making changes and updates that were needed due to changing requirements and processes. In short, the software was suffering from most of the usual failures of a waterfall approach. (Leffingwell 2007)

Using the Public Safety Development Team as an example, there are several lessons learned in how to best implement an agile development process in a non-technical organization such as a county government.

2 Software in a Non-Technical Organization

In a technically-focused organization such as a software development company or another technical company where software plays an integral role in the company's business model, the role of software is well-understood and well-supported as a factor in the business' success. At Multnomah County, software plays a background role to the county's services. For a software development team working in this environment, these differences are seen in the perception of software quality and difficulty of scheduling interaction between the business and the development staff even when such interaction is critical.

2.1 Defining Software Quality

When defining software quality, one can look at a couple broad categories. Some quality measures are absolute; these measures are often those of basic functionality. The question is "Is the software able to perform all of the functions it was designed to perform?" and the answer can be summed up as Yes or No. This level of quality is the one which is often obvious even to those outside of the industry. A second measure of quality is more relative. This form of quality includes things like the overall user experience, the ease of use, and how well the software assists the user with meeting business needs.

For external-facing software teams, this second indicator of quality can often be measured by market forces. If a team produces great applications, revenue will likely increase and customers will sing the company's praise. Buggy software or other missteps will be reflected in customer complaints across the internet and will likely result in slower sales. Customers generally have many options when choosing software and will generally select the software which provides the most functionality in a pleasant interface at a competitive price.

Internal customers at an organization such as a county government don't have such choices. At Multnomah County, if the juvenile counseling staff is unhappy with the software being produced by the Public Safety Development Team, they don't have easy options. Off-the-shelf software likely isn't an option (hence the decision to build a custom application in-house) and they can't simply find other developers to build something better. This environment requires a slightly different view to measure software quality.

Four indicators measure software quality at Multnomah County:

- Ease of Use
- Consistency with other applications
- Minimal defects post-implementation
- Perceived relationship between business staff and application development staff

Taken one at a time:

2.1.1 Ease of Use

This is simply the subjective measure of whether the business staff finds the software intuitive and is able to use the system to meet their business needs. County employees have a wide range of technical abilities; steps are taken to minimize the learning curve for users so that new employees can quickly come up to speed and training costs are minimized.

2.1.2 Consistency with Other Applications

In an ideal world, perhaps all of the needed information for a particular job would reside inside one application. The reality of working in the government is that for some function such as public safety that information is distributed amongst various organizations and departments. Depending on one's exact job duties, it's not uncommon for county staff to routinely use up to eight line of business applications and

databases to access information at the county, city, state, or national government levels. Because of the wide variety of systems used, development staff strives for consistency when building or maintaining systems so that business staff is able to easily jump between programs with a minimal amount of confusion around navigation, terminology, and data display formats.

2.1.3 Minimal Defects Post-Implementation

As is common with most development groups, the amount of defects discovered after implementation is used to gauge the quality of the development process. Many studies have shown that the cost of resolution for a post-implementation defect is significantly higher than a defect found earlier in the development cycle (Kaner, Bach, and Pettichord 2001). Defects found by customers after release also have the perception cost incurred when end users view that the software doesn't work as expected.

2.1.4 Perceived Relationship Between Business and Development Staff

This is an intangible measurement but one which has proven valuable. Because of the ongoing relationship between business users and development staff (and, as noted above, the model in which business staff is limited in application development options), there is a long-term benefit in maintaining a healthy and friendly relationship between the technical and business staff. County development staff found that perception often becomes reality; enjoying a positive working relationship with the business staff creates a positive working environment and leads to better communication (which in turn leads to better software).

2.2 Business Staff Availability for Software Development

In a technically-focused organization where software is a recognized product and/or source of revenue, organizational decisions will be made to support said software development. In an organization such as Multnomah County, organizational structure is focused around the services that the county delivers to its citizens, which isn't necessarily the ideal organizational structure to optimize internal services such as custom software development.

In the textbook version of an agile Scrum project, the software developers, customers, Scrum master, and all other stakeholders are colocated with staff sitting physically in the same location. The physical proximity makes ad hoc communication simple. One area of challenge is that the county's physical organization generally precludes having a customer (product owner) colocated with the software development team. The IT software development teams are located in a centralized county administration building whereas customers are spread out throughout the county in a location appropriate for the services provided. For the Public Safety Development team, customers are most often located at the county courthouse, jail, or juvenile services and detention facility.

Because the customer and developers are not physically colocated, it's vital that the development processes and tools facilitate easy and effective communication. For times when face to face discussion isn't feasible, the entire project team uses instant messaging for quick chats as well as the commenting and email features of Pivotal Tracker (a project management tool, discussed further below) to record decisions made about particular work items.

2.2.1 The Quest for Customer Involvement

For a member of the Public Safety development team, one's primary job function is to build and support software which helps the county conduct its business. For an internal customer of the Public Safety development team, one's primary job function has very little to do with software. These personnel are performing day to day activities that involve working with a parolee to develop job skills, assisting a juvenile offender with entry into drug counseling, or making a decision on whether a just-arrested defendant will be released on their own recognizance or held in jail until arraignment. Software supports their jobs, but software isn't their job.

The fact that software is not a primary job duty for the average county software user means that it can be challenging to develop engaged relationships between the software developers and the customers. As custom software projects begin, the reality is that the software development project will compete for the customer's time along with that individual's usual job duties. In an ideal world the customer would be able to shift usual duties onto other personnel in order to be able to fully participate with the software development team, but that isn't always practical or financially viable. With a software development project competing for a customer's time, there needs to be an explicit decision to obtain a customer's time commitment to the project.

Another challenge presented by the customer situation in a non-technically focused organization such as Multnomah County is that the customers are not necessarily savvy or skilled with current computer technologies. The county's workforce consists of individuals of all ages and skill sets, a majority of whom have working knowledge of how to use a computer for their basic job duties but with a small minority falling into the category of early adopters or those that will readily embrace new technologies. Outside of the software development teams, county staff is unfamiliar with software development practices and processes. In a software or technical organization, some familiarity with software development is part of the baseline set of knowledge. In a non-technical environment, a steeper learning curve is present for the business staff that participates in software development.

2.2.2 Customer Involvement as the Key Quality Indicator

Why point out the particular challenges with customers in a non-technically-focused organization? Because the development teams at Multnomah County have found **that the single largest indicator of the overall quality of software built in a custom development project is the involvement level of the business customer.** For projects with actively involved customers who have a decent level of technical understanding, the rate of overall project success and the quality level of the software have consistently been higher than for projects where customers have a limited level of involvement or where there are significant gaps in technical ability.

The involvement of the customer provides benefit in a few ways. With the perception of the process being a significant factor in the overall quality of the project, active customer involvement is important because the customer will feel as if they're involved in the project. Attending key project meetings (stand-ups, iteration kickoff meetings, and backlog prioritization) provides the customer with the ability for real-time input into the development process. Customers should feel that they can contact the software development staff at any point they feel it's necessary to provide feedback. From the software developers' perspective, having easy access to business staff is essential to remain on a nimble development cycle. In an agile process, working software is favored over extensive documentation. This lack of documentation occasionally results in the development staff having questions as the software is constructed. The ability for developers to access customers for rapid feedback is a key requirement for developers to move forward in a nimble fashion.

In an ideal world, development teams could pick and choose the customers of their choice. While it's not realistic to only work with customers with a high level of technical ability, the involvement and time commitment is a variable which can be controlled. On every project, the overall perception of product and process quality increased directly proportional to the amount of time involvement on the part of the business customers. When management allows for staff to spend larger quantities of time working with the software development team, the projects operate in a smoother fashion and users are more satisfied with the resulting software. Therefore it is of key importance for the IT and software development staff to convince the business' upper management of the value in dedicating business employees' time to software development projects.

3 Agile Tools: Lessons of Visibility

In moving to a more agile style of development, it quickly became obvious that new tools would be needed to facilitate agile project management and communication amongst the various project team members including the technical development staff as well as the business customers. There is no shortage of tools and processes on the market designed to make agile software development easier for an organization. Much like iterating through software features, the Public Safety Development Team at Multnomah County has used a variety of tools to support their software development, with three tools being used for extended periods of time.

No tool is perfect; each of these systems has benefits and drawbacks.

3.1 Post-It Notes on a Wall

The first tool used at Multnomah County was the most analog and had the lowest cost of entry. The large backlog of pending work was managed via a spreadsheet; at the iteration kickoff meeting these work items were transcribed onto Post-It notes. The notes were then stuck to a wall based on their position in the current development cycle. All notes would initially be in an "Unstarted Work" column, with notes being moved across to subsequent columns as the work items entered construction, then QA testing, then acceptance testing, with each work item eventually ending in an "Accepted" bucket.

Positive Results from Post-It Notes:

- The notes on a wall were very visible to the members of the development staff that sat nearby
- The act of moving a tangible item from column to column gave a sense of accomplishment
- The system required no software or other technical complexity; Post-It notes are cheap

Negative Results from Post-It Notes:

- The notes on a wall were invisible to the portion of development staff that did not sit nearby
- The notes on a wall were invisible to customers (which did not sit nearby)
- There is no easy way to archive and search the completed work items

3.2 Microsoft Team Foundation Server

With most of the group's development happening in a Microsoft .NET environment (both ASP.NET and Silverlight), Microsoft's current development tools server was a logical fit. Team Foundation Server (TFS) offers source control, a build server, work item tracking, reporting, and other tools designed to support a software development team. The TFS software runs on a local server and exposes various functions via services. Access to source control, work item tracking, build management, and reporting is obtained using client applications which communicate with the TFS services. For most developers, client access is provided using Microsoft's Visual Studio Team System Integrated Development Environment (IDE). The IDE allows for source code management and work item tracking within the same development environment used to write and test code.

For access without using Microsoft's development environment, limited functionality is provided via a web interface built using Microsoft's SharePoint functionality. Unfortunately the default views provided as part of the product did not provide meaningful "at a glance" views for business staff that allowed for sufficient visibility as to the state of the project, the work items, and a quick view as to which version of the software was ready for user acceptance testing.

This tool was in place at Multnomah County for about 18 months (two major versions of TFS). The source control and build server were used exclusively by the developers without any sort of customer interaction; from a process and quality assurance standpoint the work item tracking features were of

greatest importance. The work item features were very granular – the default forms used for recording features, bugs, and other work items featured dozens of fields. Although there were high hopes for the work item tracking abilities of TFS, the team found out that less is more; the number of fields on the screens created a barrier to entry which prevented customers from utilizing the tool. The work item tracking abilities of TFS proved to be a bit too heavy.

TFS offers the ability for customization by writing code to communicate with the server's services, but county management did not feel that spending extensive developer resources to maintain a custom toolset was a wise use of limited financial resources.

Positive Results from Team Foundation Server:

- Work item visibility via a web browser or within the IDE for developers
- Very granular, detailed work item records with many customizable fields
- Ability to record large quantities of project management data points

Negative Results from Team Foundation Server:

- Default work item views not customer-friendly; required extensive customization
- Relatively expensive server and client licensing requirements
- Requirement to maintain and upgrade web, database, and reporting servers
- Hard to evaluate and report on team strengths and weaknesses

3.3 Pivotal Tracker

Due mostly to the limitations in customer visibility of TFS work items (and ongoing licensing costs), the Pivotal Tracker tool was introduced as an alternative to TFS on an experimental basis for one project. It worked out well and is now being used across the entire development team (including customer and business analyst access).

Pivotal Tracker is a web-based, hosted work item tracking system. The vendor describes it as:

> *…a story-based project planning tool that allows teams to collaborate and react to real-world changes. It's based on agile software methods, but can be used on a wide range of projects. Tracker maintains a prioritized backlog of project deliverables, broken down into small, estimated pieces, called stories. It dynamically groups these stories into fixed segments of time, called iterations, and it predicts progress based on real historical performance (velocity).* (Pivotal Labs 2011)

Whereas TFS provided an overabundance of options for entering work items, Pivotal Tracker provides a very minimal set of fields and features. In contrast to TFS's multiple pages of dropdowns and text fields, Pivotal Tracker offers only eight fields for a work item (with half being optional). Although not as granular as TFS, the simplified interface makes it easy for developers and customers to view the important information about a work item. While the number of fields is limited, key information is captured and made visible to all. The signal-to-noise ratio is high which makes it easy to identify the current status of pending, in-progress, and completed work.

Positive Results from Pivotal Tracker:

- Good visibility to work items for customer, development staff, and others
- More affordable licensing than TFS
- Pre-built reporting for burndown and velocity is easy to use and understand
- Easy to understand "Velocity" metric which measures the team's speed at completing work

Negative Results from Pivotal Tracker:

- Custom reporting is difficult

One caution with agile development and choosing a toolset is that because agile favors communication and collaboration over documentation, the effectiveness of a given tool will often depend on the personalities and communication styles of the members of the team. In the Multnomah County scenario, Pivotal Tracker has proven to be an effective tool for tracking work items for the development staff while providing transparency for the customers. While customers and developers have differing needs from a work item tracking system, Pivotal Tracker is an ideal compromise.

One team member notes:

"TFS is a far superior project management tool because of the breadth of information it can capture. Pivotal Tracker is the best 'customer expectation management' tool." (Chapel 2011)

As noted previously, one of the quality measures is the perception of the relationship between business staff and the software development team. Managing customer expectations is a key factor in managing that customer relationship.

4 Our Nimble Solution

Given the challenges described previously and the lessons learned about tools and communications, the Public Safety Development group at Multnomah County has found a development methodology which is based on the agile manifesto but doesn't adhere strictly to the textbook definition of a particular method such as Scrum or XP. What follows is a description of this nimble development process that is working well for the technical development staff as well as the (mostly) non-technical business staff at Multnomah County.

4.1 A Prioritized Queue of Pending Work

All pending work (features, bugs, changes) is entered into Pivotal Tracker as new work items. Everyone in the organization (developers, business analysts, project managers, customer stakeholders) is empowered to add new work items. This queue of pending work is reviewed on a weekly or semi-monthly basis at a backlog review meeting. In attendance are the project manager, business analyst, business product owner, and (sometimes) the lead developer. The business product owner is the decision maker and is tasked with prioritizing the pending work.

Because the backlog of pending work items could be dozens of items, it isn't practical (or a good use of time) to review each work item at every meeting. Instead, these meetings focus on reviewing and prioritizing two groups of work.

The first set of work to be reviewed is those *items at the top of the priority list* which will become the next items to begin construction. These items are reviewed to ensure that they are well defined (the software developers have enough information to begin work) and that they are correctly prioritized.

The secondary group of items is those *recently added to the backlog* which hasn't yet been prioritized. This group will usually be a mix of bugs which have been discovered as well as new features that have come directly from the business staff or have become evident during construction and testing. Reviewing these items regularly ensures that high-priority work is properly placed at the front of the queue.

4.2 Every Other Week: Iteration Review & Kickoff

Although development doesn't wrap up in the same fashion as a Scrum sprint (more below), every two weeks the team gets together for an Iteration Review and Kickoff meeting. Everyone is in attendance: the

project manager, business analyst, business product owner, developers, and quality assurance staff. Sometimes the end user line staff is brought into the mix as well.

At this meeting, the team performs three activities. First is a retrospective of the previous iteration. Everyone is free to contribute notes about what went well along with any challenges encountered. The project manager may take away some tasks related to the challenges noted. After reviewing the previous work, the team reviews the work items at the top of the prioritized backlog. Each work item is reviewed to ensure that it is well-defined. The development and testing staff are making sure they have the information needed to build and test the work. If questions arise, the business staff and analyst may be able to help flesh out definition and resolve any issues so that the developers can begin construction. Acceptance criteria are identified. If discussion determines that an item isn't able to be quickly defined, the work item is moved lower on the priority list and will likely become part of a subsequent iteration of work.

The other activity at the kickoff meeting is noting the team strength for the coming iteration. Holidays, vacation time, or other activities that reduce staff availability must be included as a factor in order for Pivotal Tracker to establish a more accurate velocity for the team. Team strength can also be increased if there are additional resources available beyond the usual or baseline level for the project. Recording accurate team strength leads to a more accurate velocity, which in turns leads to a better ability for project managers to estimate completion of a set of work.

4.3 Allow Work to Extend Beyond Strict Two-Week Sprints

The usual definition for a sprint is that it's a fixed period of time in which a development team will commit to, build, deliver, test, and gain customer acceptance for a specific set of work items. While this scenario might work for a software company or other organization that has a focused and dedicated product owner, it hasn't proven practical for the situations described previously where a software development project is competing for a non-technical customer's time and attention. The beginning of the sprint starts as expected with everyone participating in a kickoff meeting followed by the engineering group beginning construction. As work items are completed, attempts were made to have customers test, provide feedback, and ultimately accept the work items throughout the iteration. This constant demand for customer attention was often met with failure because the business individuals did not have time to dedicate to the project on a daily basis. It proved to be near-impossible to have work items accepted on a continuous basis with all items accepted prior to the end of the iteration. Instead, work items flow into subsequent iterations if they are not completed.

4.4 Automated and Frequent Deployment Infrastructure

In order to gain rapid feedback both within the software development team as well as with business staff, easy and frequent deployment ability is crucial. The Public Safety Development Team has an automated, repeatable deployment strategy that makes it very easy to deploy the application to various development, testing, and production environments. While explanation of the entire deployment process is beyond the scope of this paper, a summary is provided for context since ease of frequent deployment has shown to be an important part of the software development strategy at Multnomah County.

4.4.1 Application Build Process

All source code is stored in a Subversion repository. TeamCity is used as the build server. A continuous integration (CI) build is automatically triggered on each source code commit in order to provide immediate feedback to the development staff if a problem exists. An integration build brings together all of the application bits, compiles them, and generates an installer (.exe) for the web application and any related services. The build also packages the database scripts to create or modify related database tables, procedures, views, and security configuration. The database scripts are zipped into another executable file.

4.4.2 Application Deployment Process

In the development and quality assurance environments, applications can be deployed with one click via the TeamCity build server. After packaging an integration build, the application is installed remotely onto a web server. The database is refreshed by starting either with an empty database (in the case of a new application) or a restored copy of the current production database (in the case of maintenance or other software updates). Database scripts are then applied via an automated job such that there is no opportunity for a flawed deployment due to human error during the deployment process. This automated deployment also helps ensure that as a set of code is promoted through the various environments (development, QA, user acceptance, and finally production) that the code is intact and is not changed as it moves from environment to environment. Basing the database refresh and deployment process on actual production data ensures that real-world scenarios are exercised and not merely dummy or test situations.

4.5 Daily Deployments to a QA Environment

Throughout the development cycle, code is deployed and tested in a QA environment on a daily basis. Rapid feedback to the developers allows for nimble development and ensures that potential problems are identified and resolved as soon as possible.

4.6 Regular Deployments to a User Acceptance Environment

Given that one of the identified keys to success for a project is the involvement of customer feedback throughout the process, regular deployments to an environment for customer acceptance are important. The exact deployment schedule is identified for a given project based on the availability and circumstances surrounding the particular customers involved, but a weekly deployment is seen as standard. After QA testing, a build is moved into the User Acceptance Testing (UAT) environment. The QA staff will notify the business staff that the software is ready for their hands-on acceptance. Users are expected to test the application, specifically looking at business functionality and the acceptance criteria which were identified at the iteration kickoff meeting.

4.6.1 User Feedback: Acceptance of Work

User feedback often occurs informally via conversations with the development team, but formal acceptance is recorded by the business staff marking a particular work item as *Accepted* in Pivotal Tracker. This indicates that the item meets the acceptance criteria that were identified and that the business believes the work item is behaving as designed.

4.6.2 User Feedback: Rejection of Changing of Work

If testing discovers problems, the business staff provides feedback to the development team. One of the challenges and training bits that happens with business customers that are new to software development surrounds identifying those issues which are bugs versus those which are changes or new features to be added.

If a work item is *broken*, that is, it isn't working as specified, then the item is marked as *Rejected* and the development staff makes changes to address the problem.

If a work item is *working as designed* but the business staff realizes that it needs changes (the classic "You built what I asked for, but not what I wanted" scenario), the item is *Accepted* and a new work item is created to reflect the changes (additional work) desired. This helps ensure that the velocity calculated by Pivotal Tracker includes credit for the original work that was requested as well as the changes.

4.7 Continuous Feedback

Although there are defined user acceptance cycles and methods, constant communication between the business staff and software developers has shown to be the most effective way for improving the software quality. This communication should flow both ways; while development staff will articulate the desire to obtain rapid feedback from the business customer, the development staff should also be providing information (and software) to the customers on a frequent basis. While tools such as Pivotal Tracker help facilitate said communication, the importance of ad hoc conversations (face to face, instant messaging, or email) cannot be discounted.

5 Nimble Development Experiences

5.1 An Estimation Solution

Software estimation is hard. Unlike assembly-line manufacturing processes, custom software development is just that: custom. Each project can often involve new business processes, new technologies, or new team dynamics. Waterfall estimation was generally a failure because in the drawn-out development process, changes of all varieties would introduce additional scope, risk, and unforeseen delays.

Despite the fact that estimation is hard and often unreliable, executive staff and those with budget concerns always desire accurate estimation.

Agile development doesn't change the difficulty of estimation. Stories are still given a size or point value. The difference is that instead of using the estimation to preemptively predict when a project will reach completion; those point values are used in a more retrospective fashion to identify the speed of the project.

Over the past three years, the development teams have settled into a project estimation and scheduling pattern that seems to both meet the desire to move in an agile fashion as well as being able to provide some rough estimates that can be used for big-picture budgeting and scheduling purposes. For a given project, the estimation and scheduling process now works as such:

1. A Business Systems Analyst (BSA) works with the business staff to gather some initial user stories and requirements. These are high-level functions desired for the new or enhanced application and would include things such as identifying the various actors, recording workflows that would become part of the application, noting any external application or data interfaces that would be needed, and identification of any potential data conversion efforts.
2. The Business Systems Analyst consults with a few members of the software development team. Based on the information gathered by the BSA and the knowledge of the software developers, a very high-level, low-confidence, low-precision estimate is made. Examples of such estimates might be "Based on other projects that seem to have similar scope, this appears to be a 6-9 month project" or "With the information we have and the unanswered questions, this seems like it would be a 12-18 month project."
3. This high level estimate is presented to the business and budgetary decision makers. Based on the information available, a decision is made whether to begin the project. There is an understanding that the initial broad estimate is just that: broad and low-confidence.
4. Development begins. Initial stories are broken down into more details so that they can be scored by the development team. Work is prioritized and the application is constructed.
5. After four or five iterations, the team's velocity (based upon story scoring and the number of points completed per iteration) becomes evident. In the meantime, the Business Systems Analyst is leading the effort to define remaining work items. These work items are prioritized by the product owner and scored by the technical staff.

6. Once a velocity develops and a bulk of the remaining work items have been scored, a tentative completion date is obtained. Although this date is not exact, it will have more precision than the previous high-level estimate.
7. Budgetary and planning decisions are made based on the new date, keeping in mind that new work will develop as the project progresses.

When resources are fixed, software development project management can use one of two targets: a date or a feature set. If one targets a feature set, development simply continues until that feature set is complete. Targeting a date, as is usually done after a reasonably accurate date has been identified, implies that project management becomes scope management. As work progresses and approaches the target date, decisions are made about which features will remain and which will be deferred to a future development project. The set of items which remains “in scope” is constantly shifting as the project team reacts to both planned work, bugs, and new work that is requested as the project progresses.

5.2 The Life of a Nimble Work Item

Software of the quality indicators, tools, and processes previously described, this describes the lifecycle of a work item in the nimble development process being used at Multnomah County.

1. Potential work items are prioritized and estimated as described in the previous section.
2. As a work item moves toward the top of the list, the business analyst ensures that background information is obtained and that the work item is ready for work.
3. At the iteration kickoff meeting, the team reviews the work item; any disagreements or uncertainty about the meaning of the work item is resolved. Everyone agrees they understand the acceptance criteria for the work item.
4. The development staff “Starts” the item in Pivotal Tracker and constructs what is needed for the work item. Developers work in communication with QA staff to ensure the design and construction meets the quality standards. When completed, the developer “Finishes” the item in Pivotal Tracker.
5. Seeing the work item is marked as Finished, QA staff deploys the relevant code to the QA Testing (QAT) environment and tests the unit of work. If rework is needed, QA staff will “Reject” the item in Pivotal Tracker and development staff will address the issues. If the work item meets the acceptance criteria, QA staff will “Deliver” the work item and promote the code to the User Acceptance Testing (UAT) environment.
6. Once a work item has been delivered, Pivotal Tracker shows it with a green “Accept” button and a Red “Reject” button. The Business Systems Analyst works with the business customer to perform user acceptance testing. Assuming things are satisfactory, the work item is marked as Accepted. If there are problems, the work item is Rejected and developers attend to the defect(s).
7. Accepted work shows in the “Done” category of Pivotal Tracker and factor’s into the team’s velocity for planning purposes based on completed work.

6 Conclusions

In a traditional software development company or other technically-focused organization, the systemic understanding of software development processes means that most of the widely-known agile development methodologies can be implemented as prescribed. A non-technical organization (such as Multnomah County) will be unable to implement the "textbook" versions of agile development, primarily due to the non-technical nature of the business customers and the challenges of obtaining active participation from customers who must juggle their software development role along with their usual business duties.

A nimble system based on an iterative development approach has proven to be successful by clearly defining the customer's role, expected meeting attendance, and time commitment for feature definition

and acceptance testing. This process has proven to increase software quality and result in a more favorable view of the software development team and process by the business staff.

References

Beck, Kent. Beedle, Mike. Bennekum, Arie van. Cockburn, Alistair. Cunningham, Ward. Fowler, Martin. Grenning, James. Highsmith, Jim. Hunt, Andrew. Jeffries, Ron. Kern, Jon. Marick, Brian. Martin, Robert C. Mellor, Steve. Schwaber, Ken. Sutherland, Jeff. Thomas, Dave. "Manifesto for Agile Software Development." http://agilemanifesto.org/

Chapel, Ed. 2011. "Why We Left Team Foundation Server" http://edchapel.tumblr.com/post/4180369020/why-we-left-team-foundation-server

Kaner, Cem. Bach, James. Pettichord, Bret. 2001. *Lessons Learned in Software Testing: A Context-Driven Approach*. New York: Wiley.

Leffingwell, Dean. 2007. *Scaling Software Agility: Best Practices for Large Enterprises*. New York: Addison-Wesley.

Pivotal Labs. 2011. "Frequently Asked Questions." https://www.pivotaltracker.com/help#whatispivotaltracker

Unusual Testing
Lessons Learned from Being a Casualty Simulation Victim

Nathalie Rooseboom de Vries van Delft
Nathalie.van.delft@capgemini.com

Abstract

Hobbies can be an inspiration for many analogies in software and system testing, but it can also be the other way around. I've been a so called casualty simulation victim for a couple of years now, playing a patient in hospitals, a victim who needs help from a first aider (both in First Aid lessons and ambulance training) and at disaster re-enactments. I used my knowledge from the Software Testing process for the benefit of being better and more structured in my casualty simulation situations. In return I got a whole bunch of tips and lessons learned that I could use within my job as software tester. Many lessons are particularly useful for software testing, but there are also lessons that are beneficial for all other disciplines in software and system development.

Biography

Nathalie Rooseboom de Vries van Delft is Community of Practice leader Testing, CTO office advisor and Managing Consultant at Capgemini Netherlands, responsible for thought leadership and testing competence development. She fulfills the roles of test manager and advisor with various clients. She speaks on national and international test events on regular basis, writes in specialist publications and participates in the Dutch Standardization Body (NEN) workgroup for Software and System development. She is very passionate about (software) testing in general, but the subjects Data Warehouse Testing, E2E-testing, Standardization, Ethics/Philosophy and Test Architecture (Framework) are most favorite.

Nathalie is also known online as 'FunTESTic'. www.funtestic.nl.

A relation between software testing and casualty simulation might be not an obvious one. But when you think a little bit further, you can see the similarities. Both are testing and although the object under test is different, the process is very similar in both disciplines. In this paper I have used a basic software testing process which is also used in casualty simulation and have extracted the lessons learned from the latter so that you might use them or be inspired to change things in your work in software testing.

1. The background of Casualty Simulation

The craft of impersonating victims has existed since the second World War. In England a gentleman called Eric Claxton started the organization 'Casualties Union' in 1942. A comprehensive history of the founding of the organization can be found on the website of the British casualties union but in short it came down to the fact the nurses and other first responders where so shocked by the (war) traumas presented to them that they weren't able to deliver help adequately. Claxton figured that if the helpers could practice with fake victims with fake traumas they could get used to it and would be able to offer the help when confronted with the gruesome reality.

Because the British initiative proved to be successful also a Dutch version of the union was started, called LOTUS (Dutch abbreviation for Education for Impersonating Casualty Simulation Victims). Every LOTUS, as every casualty simulation victim is called, has a certificate 'First Aid' and follows a two year course, which is finished by a practical exam. In contrast to what many think in software testing, domain knowledge is crucial for executing a scenario successfully and a casualty simulation victim is taught this domain knowledge, sometimes with specialties, extensively. A casualty simulation victim is obligated to go to a minimal number of lessons each year to keep the LOTUS certificate (and do a practical competence test). Practice is just as important as knowledge.

2. A familiar testing process applied in an unfamiliar environment…

Within my casualty simulation work I use a familiar testing process. I specifically say A testing process here and not THE testing process, because there are of course more processes, but this process works best for me in the casualty simulation situations. I use the process as is shown in figure 1.

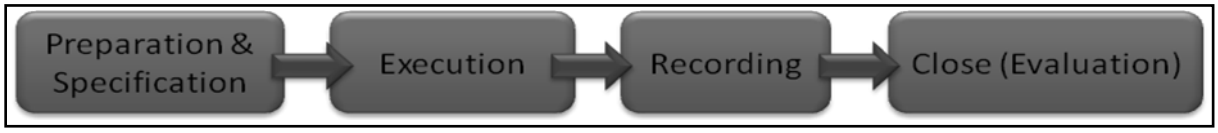


Figure 1, a familiar testing process…

2.1 Preparation & Specification

Just like in software testing the preparation and specification phase is the phase where you set your goals. You can perhaps imagine that a First Aid lesson has a whole other goal, namely the education of a person, than a drill in a hospital or a huge disaster re-enactment, where the medical staff is supposed to give the right medical treatment.

The specification gives the story in rough lines. As casualty simulation victim you get a request for a drill, exam or lesson. In this request is described what the event is that happened, in case of a drill, or what the theme is in case of a lesson. In a complete request there is also a description of the trauma. After that you decide how you're going to moulage this (moulage is the term to put on the make-up). You can do extra research to how the trauma is supposed to look and what the behavior is of a patient that suffers from that particular trauma. I use medical databases like PubMed for this purpose. Finally you decide what your story is. A good first responder will always ask what happened, so you'll have to have a story that fits (except when you play a confused victim of course).

In my work as software tester this helped me to realize detailed scenarios for end-to-end testing. In my previous project I tested purchase and validation systems. By making a very detailed story about what type of customer I could be, as close to real life as possible, I can also test the system as a real life person would use the system.

***TIP**: In your organization it might be beneficial to visit the marketing department. The department ought to have customer profiles that you can use to create 'mindsets' that you can then use to make your stories to test.*

Preparing moulage for disaster re-enactment Livex 2009.

For a casualty simulation victim it is really important that the environment or scene is correct. The environment has to tell the story to the first responder so that he, or she, can determine a diagnosis with help of this environment. An unconscious person next to an exposed power cable can give a whole other diagnosis as the same person next to a bottle of chemicals, or even without attributes the diagnosis is different again. Also in software testing it is important to have your environment in order and to have it very clear what this environment is when testing is done or used in production. A specific defect can have a whole other meaning in one specific environment than in another environment.

2.2 Execution

After the preparation and specification phase the execution phase starts. For a Casualty Simulation Victim, this means playing your role and continue playing that role. A first aider (or even doctor) knows that it's a drill, but if the casualty simulation victim really plays his, or her, role well and with conviction the first responder will react more naturally. I even have an anecdote about where the first responder forgot that it was a drill…

Every week I play the role of a woman in labour distress in an Amsterdam Hospital. In this hospital the delivery room trains with weekly scenarios where an emergency situation emerges during a normal labour. The obstetrical team has to solve the problem with the means they have available. The situation is made as real as possible, we use a real delivery room, use real medication (not really injecting it ☺) and have to wait real-time waits for results. I have all the IV's and monitoring devices attached (with tape) to my body. Although the team KNOWS it's a drill they DO react as in real life. At one time there was this assistant gynecologist who tried to pull off my specialized pants (which simulate certain clinical picture in

obstetrics) to be able to do an internal exam. Luckily my costume was well secured to my body and the training leader was well in time to stop him.

My experience is that the more real life you make the scenario, the more detailed and accurate the information obtained from the scenarios will be.

***TIP**: Sometimes it helps to re-create the environment that your end-users will have when using the system, including time-margins and noises. Environmental factors can have an impact on usage of a system that isn't always taken into consideration testing in your comfy office…*

Photo: Jorit Schlenter, Livex 2009

At the execution phase the factor FUN is an important one. The more fun you have during the execution, the better you will do your job. Mostly you play misery, fear and pain, but if you feel really miserable doing this, your role – strangely enough- will not be convincing. If you have fun playing the role the results are accordingly so. You will be much more convincing!

Still, playing a casualty simulation victim isn't without any danger. The first rule we learn in our casualty simulation lessons and in the LOTUS learning book is: "Mind danger and take care of your own safety". I can give an example of a casualty simulation victim where the first aider started the reanimation procedure and broke a rib. That should never have happened. The same goes for the 'Heimlich' maneuver when playing a choking scenario, I always place my hands between my body and the hands of the first aider and tell them: "Ok, stop now and show me gently where you would place your hands and how you would make the maneuver when using it forcefully".

If a real emergency occurs we know the stop word "NO PLAY". This is an internationally acknowledged sign that the drill has to be stopped immediately. In a large drill re-enactment there was a case of real hypothermia victims at a certain point. Because we had casualty simulation victims playing hypothermia too, it was impossible for the first responders to spot the real victims. For a certain time the drill was stopped by calling the NO PLAY situation. The real hypothermia victims could now be localized very fast and helped. It works the other way around too. When you play an unconscious victim and you really fall unconscious, the first aider can ask you 'No Play?'. You are obligated to react on this question; when you don't react the first responder will know you are a real victim and will act on this, by calling 911 for real for example.

I once used the ‘no play’ rule in my work as software tester, which a introduced to the participants of the project in an earlier stage with the term ‘no test’. During an end-to-end test I ran into an issue that would also would be highly likely to occur in the (current) production systems and had high priority. I used the ‘no play’ situation and by doing so the organization knew it was a serious situation and acted upon this accordingly. This resulted in a joined effort where the issue was solved within hours (mind: this was including pinpointing, risk analysis, data adjustment, re-testing (including scoped regression test) and roll-out on production systems) . The organization knew that with this ‘no test situation’ it was a real serious situation and also gave commitment for acute assistance from people in their departments. It’s of course very important that these kinds of agreements are clearly communicated throughout the whole organization AND that it is not to be used lightly. A ‘no test’ situation decision should never be taken ‘alone’ but only after analysis so that you can do this with a certain amount of certainty AND only then in collaboration with management.

Exploded firework in hand, moulage from lesson ‘firework trauma’

2.3 Recording

Also in casualty simulation work there is a recording phase. This is somewhat different than in software testing though, as you cannot bring any notebooks or take notes when playing a role. So you just have to remember everything. Luckily there’s a nuance which narrows ‘everything’ a bit down; you have to remember everything with the goal in mind. I do this by using keywords. Because I have prepared the scenario very well, I only have to add the bits and pieces to the scenario I have memorized. With huge drills there are photographers, but they only take atmospheric images and one cannot rely on those pictures to reconstruct the problem or learning from them.

At specific exercises or drills there is a team of observers. For example in my hospital drills there is an observer for each discipline that participates in the drill, examining the medical skills of each discipline, including the timeline. At First Aid exams the observation is done by the examiner. In rare occasions video is used. I find this a very powerful tool, because you can rerun the tape over and over again and details can be highlighted even more.

External observation is very powerful. This is also the case in software testing. This is one of the reasons that where techniques like “pair testing” are used or where one tests in teams are so successful and give relatively good results. I’m also a huge fan of usability labs, where a user can be filmed during the usage of the software. Sometimes a very small remark can give a lot of information that would have gotten lost in an unobserved test. I also find observation during regular processes very useful for the development of (specifically end-to-end) test scenarios . By observing the user in his regular working process performing

his tasks you can prevent design of tests where your user later remarks “Oh, but we don’t ever do this in real life”.

I also found that observing a user adds a bonus. The user feels that interest is taken in his or her job. This results in an increased involvement from the business in the (testing)project, contributes to the team bonding and helps with the acceptation of the product just because the attitude is more positive as a whole.

***TIP**: You can use your web-cam as a recording device during your execution to film your responses and exclamations during tests. It will sometimes give you unique insights and a whole different perspective on your execution.*

2.4 Close [Evaluation]

In software testing, the close is probably the most neglected part of the process. The focus mostly lies on fixing (as fast as possible) defects, re-test and delivery of the product. In casualty simulation the evaluation takes up the same amount of time (generally) as the whole execution and there is a lot of information and lessons learned in the process.

After the drill is finished the evaluation is done. This evaluation takes up at least thirty minutes. The evaluation is divided in two parts. First the observation team does an internal evaluation, without the participants. Of course the participants mostly discuss vigorously in the hallway about the drill. The second part is done with observers and participants. They always start with the good points and what went right. After that the learning points are discussed. The evaluation is always done with respect and understanding to each other’s knowledge, abilities and perception.

Drill in hospital, re-enactment of labour distress; pre-eclamptia (seizure due to high blood pressure)
Photo: Joost van de Broek, Volkskrant

What I noticed particularly is that during these evaluations one never points a finger to a specific party or participant, never blames a person. Maybe this is because in these drills they are focused on the positive feedback. One always gives feedback and suggestions in a constructive manner. One simply doesn’t seem to be busy with who is responsible for the mistake or error, but one makes the observation that something went wrong and after that directly focuses on how to cope with that and how to solve it.

I notice myself that the 'finger pointing' is a difficult point within my job as a software tester. One seems to think that by figuring out who is to blame, who is the guilty one, that one can find the solution there. This results in a vicious circle. Nobody is willing to work together anymore when one gets the blame and one will take a defensive attitude where he/she will try to shift the blame to another person. Meetings get a very tense character. It is very important to focus on the positive things when starting these kinds of meetings. After that, one should very factually sum up the problem(s) without any judgment and one should emphasize that a solution is to be found. Every attempt of 'finger pointing' should be resolutely stopped.

***TIP**: In meetings where everybody seems to speak at once and there's no structure or willingness to hear each other out, you can use the 'speaking attribute' (also sometimes referred to as 'talking stick'). It's a small object that gives the holder the 'speaking right'; everybody who doesn't hold it MUST be quiet. It seems a childish solution, but in my experience it apparently helps to have a physical object making it visible who speaks.*

Another lesson from these evaluations is 'letting it land'. In most cases somebody says something and, without considering it thoughtfully, very quickly the next point is addressed. How many times did it happen that you say something to somebody and later on that person says he doesn't remember you saying that? It's important that you verify that what you say has 'landed' at the recipient. You can do this by asking the recipient to summarize what you said or let them give an active reaction. You can insert a pause between points and give the participants in the meeting some time to think about what is said and to react. In my opinion this is done way too little and valuable time is lost: firstly because people forget what you have said and secondly because you have to keep repeating your message.

3. Four Gems: Added value, jargon, checklisting and SpeakUp!

Doing casualty simulation drills gives a huge amount of information, tools, thoughts and small, quick solutions you can use in software testing. I will share four of these jewels in the next part of this paper.

3.1 Added Value

Added value is something we Dutch know from the fishing business. One fishes for sole, but one catches all other kinds of fishes you can eat too. During one of my casualty simulation drills in the hospital I got severe blood loss on the 'first line' of help. The bleeding was of such a serious nature that I had to be transferred to the 'second line'. During this drill (in the evaluation) a participant noticed that finding the arteries at the second line locations would be very difficult because pressure would have been almost gone. In some cases the anesthetist had to be called in to set an IV. Nowadays the rooms in the first line have an IV set in their stock and the nurses there have learned how to apply an IV. Valuable time is won because one can administer fluids even before transfer to the second line and the survival ratio of patients has gone up.

In software testing you can also have added value, but most of the time this isn't noticed, is considered non-relevant or not in scope. By actively noticing these gems and returning the added value information to the organization, you can be an added value yourself to this organization you test for. It's still up to your stakeholder to do something with it, but not mentioned at all, is indeed a missed opportunity for certain.

***TIP**: As testers we use the system more extensively than a normal user would and we can notice Repetitive strain injury-invoking (RSI) movements (for example clicking on a sequence of buttons) much faster. You can use this knowledge in your advice to your stakeholders.*

‘model picture’ during one of my moulage lessons, Photo: Erik Jonker, 2009.

3.2 Jargon

From one of my larger drills that I participated in (disaster re-enactment of the Dutch Flooding Disaster of 1953, called FloodEx) to practice the dispatch of incoming foreign help units, I got a DVD with the evaluation. It’s worth its weight in gold ten times over. A number of quotes from this DVD are:

“Another major issue is ‘language’, ’terminology’ and ‘jargon’. Not all participants speak English and not all those who claim they do – for example the Dutch- speak it in such a way that the English would necessarily recognize it”

“…when conversational language was not an issue, the use of jargon and acronyms were. There was a tendency to assume that the foreign units understood a jargon that the Dutch believed to be universal but that was in fact very much their own” and

“One cannot expect the Dutch civilians being saved, to understand ‘Latvian’ or ‘Polish’…”.

All three these quotes are connected to communication. Also in software testing we know this issue. As software testers we use a jargon that is perceived by us as normal language that is understood by everybody who we talk with. We simply don’t think about it that a non-tester doesn’t understand our language at all. The same also counts the other way around. An accountant who has to explain what he means, uses his jargon unintentionally, but it’s very difficult for us testers to understand. It’s not a requirement to understand each other’s language to be able to do a good testing job, but it IS very useful to have a translator present who is really into both jargons and can take some distance from the matter involved to avoid misunderstandings and wrong assumptions. This was also one of the tips from FloodEx; by using translators, valuable information was not lost.

3.3 Checklisting

In the business of first aid checklists are used extensively . Particularly in stressful situations a checklist provides a grip so that (fatal) mistakes can be prevented.

In a certain hospital the side on which an operation was to be done was switched. So instead of operating on the right side, one operated on the left side and vice versa. In this hospital the checklist 'TOP' (Time out Protocol) was introduced. It's a list with questions that is used during transfers of the patient from one department to another, specifically before a patient is going under anesthesia. The list is run through in the presence of the conscious patient. Running though this checklist only takes a minute. That minute investment saved a lot of misery in the future. When the hospital implemented this TOP checklist only one switch of sides on which to operate, occurred in two years. Investigation showed that in that case the checklist wasn't used.

Also in the hospital where I drill, they use checklists. They have checklists for every acute situation. During these situations these checklists are run through, even when time is critical. Forgetting something can be a lot more fatal then an extra minute that it costs to run through the checklist.

In software testing I once in a while see a checklist passing by, but in my opinion the added value of a checklist is highly underestimated and is used way too little.

It seems that using a checklist isn't 'high tech' enough and is perceived as inferior material. Specifically during transfers from one team to another team a simple checklist with items to discuss can have a priceless value, one doesn't have to remember every item, which probably cannot be remembered anyway. The checklist should be a standard (help) tool in the whole software development lifecycle.

The time-out protocol from the hospital (taken from the book of "M. Heres, Simulation Training..")…

Name of patient
Date of birth

Medical indication
Are there allergies?
Which medication
Is there maternal disease? Time of pregnancy
Duration of pregnancy

A translation to a time-out protocol for software testing [example]…

Name of system
Version number of system
Date of release
Known issues
Specific workarounds
Are there known difficulties in the design or build?
Time till next release

3.4 Speak Up!

The fourth and also last lesson in this article is ‘Speak Up!’. For us Dutch it’s very common to speak our minds if we think something is fishy. We are proud of this ‘feature’ that seems to be typically Dutch. But even in our country it isn’t customary for everybody to say everything to everybody, specifically in hierarchal organizations or structures, and this is even more the case in countries where hierarchy is more embedded in the cultural background.

A notorious example in history is the plane crash in Tenerife in 1977. A Pan-Am aircraft landed on a KLM aircraft on the runway. There were 583 casualties to mourn. This disaster might have been prevented. Investigation showed that the pilot, a highly decorated pilot veteran was irritated by the long waits at the airport and was anxious to leave. Beside him was his co-pilot, who respected his boss very much, saw an error to be made but didn’t say anything because he was confident the pilot knew what he was doing. If the co-pilot had used ‘Speak Up!’ on that particular moment the pilot could have noticed the error and it probably wouldn’t have occurred at all.

SpeakUp! is a handy tool where one speaks out loud what one sees, asks a question or makes a remark. The recipient can react to this by, for example, saying that it is a conscious choice or can adjust his actions when an error is indeed in place.

“Fore?? Where??” a small golf ball will cause a massive wound on the head and leave one disorientated. Lesson ‘first aid at sport injuries’

During one of my first drills a gynecologist wanted to administer a medicine where he erred in dosage. In the scenario I played that I became worse instead of better, only because of quick responses and giving me the antidote I was saved. In the evaluation a student who participated in the drill noted that she had seen the error made but was afraid to say something to the highest ranking doctor in the room. The gynecologist reacted that he had made the error because he had had a really bad morning that day and that his thoughts weren’t at the drill a full hundred percent. He would have really appreciated that the student had said he made an error before giving the medication. A few weeks later a similar scenario was executed. There was a different doctor, but the student was there too. This time a medication protocol was used that wasn’t applicable to the trauma played. The student used ‘SpeakUp!’, spoke out about the thing she saw and asked the question about the protocol. The doctor thanked the student and

adjusted his action. He had reacted on instinct, using a method he always used, but the newest insights and developments weren't customary for him yet. The SpeakUp! method had helped that a better, and for the patient less invasive medicine was used.

SpeakUp! is also very applicable in any team in software development. I would like to add a bit more in this case by making this principle mandatory or giving a bonus when using the SpeakUp! technique. Simply stating "You can say anything in this team when something is up" won't work. The more present a hierarchy in an organization, the less people will use 'SpeakUp!'. By making it an obligation and even adding a reward it takes on a whole other character and one will see it as a "must use". As long as the 'SpeakUp' technique isn't common practice, this is the way to stimulate it. Also in evaluations the SpeakUp! usage should be actively discussed and should be a reoccurring item on the agenda.

4. For closure

Inspiration and passion from a hobby can give a great contribution in your work, but also the other way around. I experienced that having passion for a hobby radiates to your work and vice versa. And by applying the lessons learned from the one, the other has benefitted from that. Whether I use a software testing process in casualty simulation or using the evaluation lessons from casualty simulation in software testing.

In this paper I have shown you that a familiar testing process can be applied in an unfamiliar testing environment, like in casualty simulation. In doing so, I got to extract lessons learned from each phase in this process that I could in turn use in my work as software testing.

A quote from the FloodEx drill stated: "We learn much more from the few things that go wrong than from the many that pass without a glitch".

I feel that as a software tester it is important not to only look at these many things that go wrong and learn from them, but also look at the positive things we can offer an organization. By applying (one of) the four gems like 'SpeakUp!', 'added value', 'jargon' and 'checklisting' we can offer a positive contribution to our work and the organization we work for without too much investment. In my paper I have described both the casualty simulation situations as the experiences with the gems in my work as software tester, so that you have an idea how to apply the gems in your own working place.

And the most important thing is that you have fun in whatever you do!

References

Braxton, E. (1942) Casualties Union, http://www.casualtiesunion.org.uk/

Heres, M & Vermeulen H. (2010); Simulatie Teamtraining Acute Gezondheidszorg en Verloskunde, leer- en werkboek, Garant-Uitgevers, ISBN 978-90-441-2527-6 [Dutch]
(translation:) Heres, M & Vermeulen H. (2010); simulation teamtraining, acute healthcare and obstetrics, learn- and workbook…

Het oranje kruis, LOTUS leerboek, 2005, Drukkerij Groen [Dutch]
(translation:) The orange cross, LOTUS learning book, 2005, Publisher Groen.

More information

Dutch LOTUS organization: : www.organisatielotus.nl (Dutch)

UK Casualties Union : www.casualtiesunion.org.uk (English)

German casualties organization : http://rud-team.de/ (German)

Belgium casualties organization : http://www.hvk.be/venus_westvlaanderen/ (Dutch)

Training US (combat simulation) : http://www.militarymoulage.com/ (English)

Photographs

Jorrit Schlenter, photo of Livex 2009 drill
Joost van den Broek, Volkskrant, 2010, photo of hospital drill, pre-eclamptia
Erik Jonker, 2009, Photo of LOTUS lesson 'model photo' (wound on arm)
Other photographs: personal archive of Nathalie Rooseboom de Vries van Delft

Kanban – What Is It and Why Should I Care?

Landon Reese
Kathy Iberle

Abstract

Kanban is gaining popularity in the software development world. It deserves to be considered as a means to manage software development. Kanban is a lightweight agile model which provides visibility to work in process, the capacity of a given resource pool, and the current workflow. The Core Test Strategy Lab at Hewlett Packard has adopted Kanban, and has seen concrete evidence of its efficacy. Using our experience in the Core Test Strategy Lab, this paper will do the following:

- Explain how Kanban can be used as an agile software development method
- Provide some guidelines for running Kanban effectively
- Lay out some situations where the use of Kanban will be of benefit
- Elaborate on how the low upfront cost of Kanban accelerated its adoption
- Identify some situations in which Kanban *will not* be of benefit

Biography

Landon Reese is currently a project manager at Hewlett-Packard in the Core Test Strategy Lab. Within his current role he has implemented a Kanban process to manage tool development, and the build of a data center at the Boise site. Prior to his current role, Landon spent two years as a firmware engineer on enterprise class laser printers, responsible for scan ASIC turn-on and testing of the scanner interface.

Landon has a B.S. in Electrical Engineering from Santa Clara University.

Kathy Iberle is a senior software quality engineer at Hewlett-Packard, currently working at the HP site in Boise, Idaho. Over the past twenty-five years, she has been involved in software development and testing for products ranging from medical test result management systems to printer drivers to Internet applications. Kathy has worked extensively on training new test engineers, researching appropriate development and test methodologies for different situations, and developing processes for effective and efficient requirements management and software testing.

Kathy has an M.S. in Computer Science from the University of Washington and an excessive collection of degrees in Chemistry from the University of Washington and the University of Michigan.

First published at the Pacific Northwest Software Quality Conference 2011

1 Introduction

Kanban is a fairly new agile software development method, derived from Lean methods. Like all agile methods, Kanban insists that work must be split into chunks which have value to an end user or buyer. However, Kanban manages the chunks of work differently than many agile methods:

- Kanban doesn't insist on cross-functional teams.
- Kanban doesn't prescribe specific roles for the team members.
- Kanban uses a "pull" system which can operate with *or without* fixed iterations or sprints.

This makes Kanban work well in some situations where it's hard to see how to apply Scrum and other popular agile methods.

Our organization has had considerable success in applying Kanban. This is the story of one of those efforts.

2 What is Kanban and how does it work?

Kanban is a system for optimizing both the number of work items which can be done by a given team and the response time to new requests. The method was developed in Japan in the 1950s to maximize manufacturing system throughput. The method is called "Kanban", which means "billboard" or "card" in Japanese, because physical cards are used in many Kanban systems. Kanban and related "Lean" methods were later demonstrated to work on any system consisting of discrete items (such as a feature), activity states (such as development or testing) and wait states or queues (such as "waiting to be tested").

Kanban makes several assumptions:

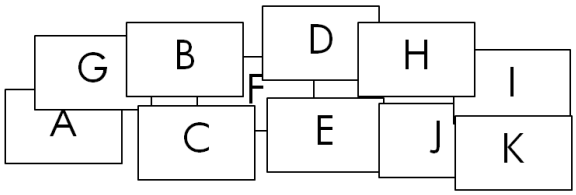
- There's a bunch of work to be done which can be split into discrete work items.
- The work items tend to arrive at different times.
- The team doing the work does not have infinite capacity.
- People get more work done when they can focus on one or a few items, rather than bouncing between many different items in the same day.

Agile software development fits into this model quite well – our work items are generally features, the customers persist in thinking up new features, we cannot do an infinite number of features, and we definitely get more done when we can focus.

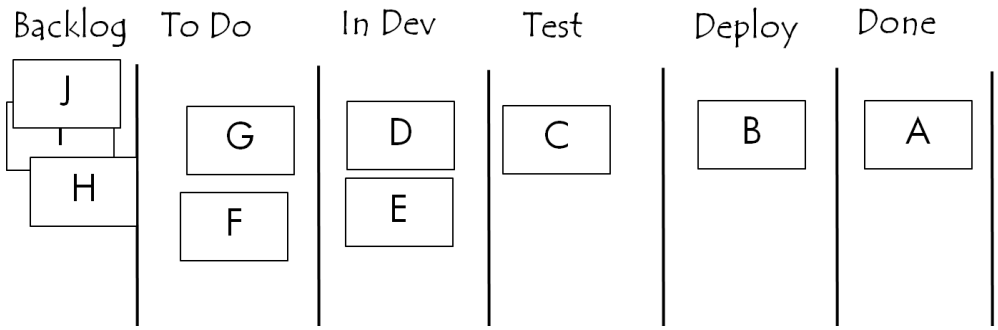
Why Kanban works can be most thoroughly understood by using queuing theory, which is the mathematical study of items moving through activity states and wait states. [REI2009]. Queuing theory itself is beyond the scope of this paper, but it is possible to understand most of the effects by envisioning your system as a set of states through which the work items are moving. In this paper, we'll use these ideas to show how Kanban focuses on maximizing the throughput of work items through a software development system, and then give concrete examples of how we used Kanban and what we discovered.

First, let's look at how Kanban allows us to focus while maintaining a predictable and speedy response time.

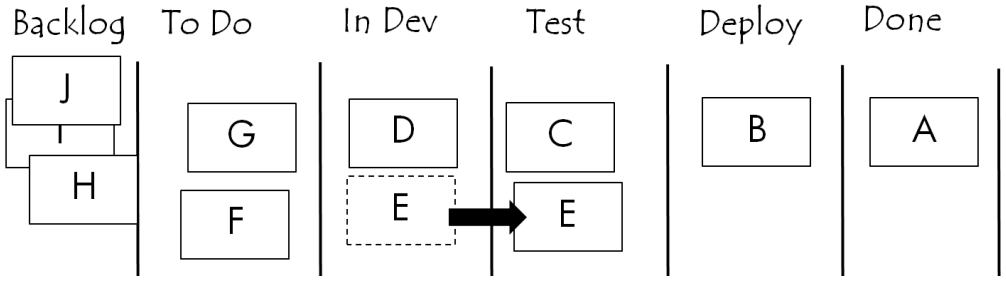
In Kanban, the work is managed this way:
1) The work is split into pieces, which are represented by cards.



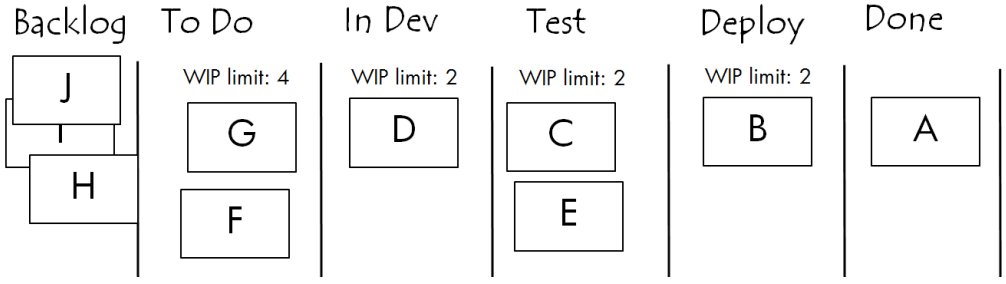
2) The progress of the work is visually tracked through the necessary steps or states, using the cards. This display is visible to all team members (and any other interested parties) at any time.



3) Work is "pulled" through the system – that is, new work items are pulled into each step by the people who are responsible for that step.



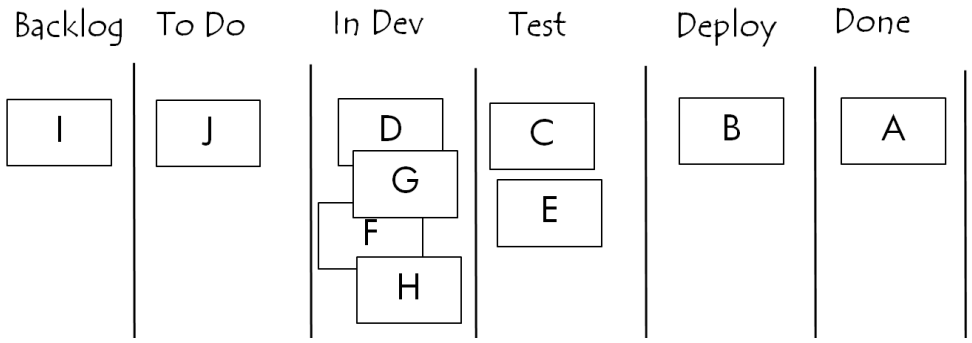
3) The flow of work through the system is controlled by strictly limiting how many work items can be in each step at the same time. This limit is known as the *Work-in-Progress* or *WIP* limit. New work items can be "pulled" only if the step is below its WIP limit. In the situation below, "D" cannot be pulled into testing because Test is already at its WIP limit.



This likely will leave one of the developers idle. Instead of pulling one of the "to-do" items into development, the developer is expected to go help the test team until the number of items in Test drops below the WIP limit.

So, why does this create a predictable and fast response time? There are several reasons:

- The strict WIP limits allow the team members to focus on just a few work items at a time. This allows the same number of people to do more work (with less stress) because they aren't wasting time on task-switching.
- The progress of work is extremely visible. If one step has gotten "stuck", it will be obvious to the team in two ways: the step has stopped pulling items from the previous step, and the next step will not be able to pull any new items because nothing is ready to be pulled. People in both the previous step and the next step will quickly run out of work.
- The WIP limit forces the team to fix the "stuck" spot, instead of continuing to pile up work in the system. Without a WIP limit, if testing was stuck, development would continue to take in new work, resulting in a situation like this:



The developers are busy, but no new features are getting to deployment because the test group is overloaded. More development will not fix the problem – more testing is needed.

In Kanban, if one step is at its WIP limit and the upstream step has finished one of its items, the item cannot be pulled into the "full" step. Instead, the upstream people are expected to go help with the step that is at its limit. In this case, the developers are expected to go help the test group until a spot opens up in the test queue.

This seems counter-intuitive to many people, because the developers will (usually) not be able to test as fast as the testers, not being as familiar with the tooling and so forth. However, the overall throughput of the *entire system* is higher – even though some individual people aren't working at their personal top speed. This is provable using queuing theory, and has been demonstrated in practice in manufacturing repeatedly for the last few decades.

There's a more detailed set of pictures of how Kanban works in [KNI2009], which is readily available on the web.

3 A more detailed look at Kanban

The first step in Kanban is to define work items or chunks of work. The same rules as used in agile development apply, but envisioning your development system as a set of states does make some things a bit easier to decide.

The first issue to consider is the type of items which can be treated as a chunk. The point of Kanban is to optimize the system to produce saleable stuff, not just do work, so the work items *must* have value to an end user. Chunks can be stories, minimum-marketable-features or anything of that sort. Architecture documents, investigations, and partially finished code are not considered legitimate chunks.

The second issue to consider is the ideal size of a chunk. The chunk is acting as a batch moving through the system. If the chunks are too small, the overhead of dealing with each chunk (for instance, entering it in the Kanban system, and prioritizing it) becomes too large a fraction of the total work being done. If the chunks are too large, the response time to customer requests becomes very long. (For more information, see the discussion of "transport cost" and "holding cost" in [REI2009]). Fortunately, it's not necessary to hit the exact ideal batch size to get fast, predictable throughput. Most organizations try out some sizes until they are getting acceptable throughput.

In Kanban, the chunks start out on a backlog. This backlog isn't prioritized in a 1 to N order. Instead, a group of stakeholders meets regularly to decide which chunks should be started and those are marked as the next ones to be pulled.

Once a chunk is "in progress", it moves through whatever states are needed to accomplish the work in this particular process. The team defines the states based on its normal workflow and handoffs between different people. For instance, if one group of people develops a change and a different group deploys the change into production, there would be two states – "Develop" and "Deploy". The team maintains a *Kanban board* showing, for each chunk, its state and who is working on it. Co-located teams usually have a physical board with Post-it® notes for each chunk, while geographically dispersed teams need an electronic board. (See Appendix A for a sample Kanban board.)

As described earlier, WIP limits are maintained on each state as well as the entire board. The developers aren't allowed to push a story into deployment if the deployment group is already at their WIP limit. Since they can't push a story into deployment, they also can't pull another story off of the backlog and start it, because that would overrun the development WIP limit. The development team's only remaining option is to go help the deployment team clear their queue, which is known as "swarming". The result is that the flow of work through the system is maximized – the largest possible number of chunks is accomplished in a given time box.

As in Scrum, there is a daily standup meeting. In Kanban standups, instead of querying each team member about what they are doing, the team "walks the Kanban board". Starting from the state closest to "done", the team looks at each chunk, asking "is this progressing as expected? Does anyone need help to get this chunk done?". There are no progress reports, because the Kanban board is visibly showing the progress.

4 Our experiences with Kanban

4.1 Who we are and what do we do

The Core Test Strategy Lab (CTSL) is a system integration and test lab for LaserJet enterprise systems. Hewlett-Packard (HP) releases a large number of LaserJet products each year. These products consist of a great deal of sophisticated software, firmware, hardware, and allied web services, which are produced by dozens of individual teams. The Core Test Strategy Lab provides system integration and testing at the system level for these solutions. The lab altogether comprises around 100 people.

We started managing all our work in Lean fashion January, 2010, and three of our teams adopted Kanban in August, 2010. The three teams are respectively

- designing and developing reusable tests
- creating and maintaining a bevy of small tools for use within our lab and maintaining our QualityCenter projects
- designing, implementing, and executing very large-scale tests in our Enterprise Test Lab.

4.2 The Tools Team Learns Kanban

In the summer of 2010, the CTSL Tools Team was searching for a means to accelerate their work. High demand from several stakeholders made prioritization difficult. In order to get work serviced, stakeholders began requesting work at the last minute and setting near impossible delivery dates. Several requests were given to the team on the same day the deliverables were needed. Resources felt overloaded and unable to get enough time to deliver key enhancements to the organization. Due to the lack of time, plenty of technical debt had been inserted into the tooling, making small updates and maintenance continually take more effort than expected.

Kanban was presented as a possible solution to the tools team by Kathy Iberle. She learned of its uses at the 2010 Lean Software and Systems Conference[1] , and afterwards introduced it to the lab. We ordered a copy of David Anderson's book, *Kanban: Successful Evolutionary Change for Your Technology Business* [AND2010], for each member of the tools team and required them to read the first several chapters. Then we decided to try Kanban, but we needed to decide how to implement it. The tools team is remote, so each member was flown in to Boise, Idaho to discuss the process. An entire day of the offsite was dedicated to Kanban training, led by the manager of that team.

After the day of theoretical training and answering concerns, the tools team discussed how they would implement the process to begin a pilot. The team started creating its own Kanban board by deciding on WIP limits for each column. This decision is an important one for Kanban. Setting the number of slots is Kanban's method for reducing WIP and task switching. Any columns with too many slots create opportunities for work to pile up and not get completed. It also eliminates the ability for teams to collaborate and work together on projects. The tools team purposefully dropped our limit such that every person could not have more than two independent chunks of work in the system at the same time. We decided to have a WIP limit of ten in the under development state and committed to a WIP limit of seven in the committed/input queue. This queue would replenish weekly in a joint meeting with stakeholders,

[1] The LSSE conference is produced by the Lean Software and Systems Consortium http://www.leanssc.org/

where the stakeholders advocate amongst themselves on which requests should be committed to each week.

4.3 Day to Day Kanban

In order to understand the benefits of using Kanban in the tools team, it is important to understand how the daily activities of a developer on the team are driven by the process. At least three times a week the day begins with a standup meeting where the team reviews the Kanban board. The board is covered from right to left, first discussing Kanban cards in the done state, then under development, and then finally what is in the selected queue. Work is discussed that has been stuck in a state for long periods of time, looking for opportunities to swarm and briefly brainstorming on possible solutions. Unless work is stuck in a state, current status is not shared.

There is no time spent on Kanban cards in backlog. Only work that has been committed to gains dedicated resources. If a developer completes one requirement, the developer checks first with any blocked team members if they need assistance before pulling from the committed queue. This drives focus and attention on finishing work before starting new work.

4.4 How work gets prioritized

At any given time, we're serving half a dozen different programs, each with its own stakeholders. We use Kanban's method of prioritizing the work at an Input Queue Replenishment meeting. This gives all the stakeholders a voice in what to do next, in a quick and organized fashion. As the managers of the teams use Kanban, they learn the capacity of their team (similar to agile "velocity") and can avoid overloading the team, which would cause slow response time.

The Input Queue Replenishment meeting ensures that each requirement coming into the tools team is agreed upon as a top priority by all necessary stakeholders. This required the stakeholders to decide which problems to focus on, which was an excellent means to limit work in process for the tools team. Each item placed in the queue has an agreed-upon value, timeliness in its need for delivery, and an actual or surrogate end user ready to verify and provide more specifications to the tool developer. If there wasn't a consensus at the Input Queue Replenishment meeting or there wasn't a solid agreement, the work item doesn't even make it into the "committed" queue.

4.5 The Kanban Board

Because all our teams are geographically dispersed, we're using electronic Kanban boards. We are tracking the work chunks as QualityCenter "requirements" and using a custom-written Excel add-in to display the Kanban board (See Appendix A). There are numerous Kanban-board tools available commercially today, but our IT department is not supplying any of them as of this writing. (Many of them store the data outside HP firewall, which disqualifies them for obvious reasons). It is possible to manage a Kanban project directly from QualityCenter, by representing all the chunks as requirements, but displaying the information in the classic Kanban board format certainly makes it easier to see what is going on.

Most authors recommend that the Kanban board be publicly visible on a wall, in the style of agile "information radiators". We can see the value of having our progress visible to our stakeholders all the time, but we haven't yet developed the technology to make this possible.

4.6 Benefits of Kanban

The tools team saw immediate benefit from the shift in day to day activities during the Kanban pilot. The WIP limits per queue allowed for strict capacity control without discouraging collaboration. The Kanban board itself brought to light the length of time relatively small changes were taking, driving buy-in from stakeholders to allow the tools team to pay some of its technical debt. The weekly meeting to insert work into the team's "committed" queue eliminated the last-minute requests. Once the maintenance and minor changes were in control, the team was able to begin addressing new feature requests. The developers did not feel overwhelmed or overworked, since regardless of the backlog size only a certain amount of work was going to be committed to each week. The turnaround was noticeable from stakeholders as well as developers in just one quarter, and afterwards other teams in the lab were looking to implement Kanban to reap the benefits.

We found that just having a Kanban board has returned time to the developers. It has removed the constant "status checking" from managers for the developer, as well as the lag time in getting feedback for the manager. Any stakeholder, partner, manager or interested individual can instantly get an understanding of the team's scope of work and what it currently has in its system by checking the Kanban board. There is a reduced need to request status e-mails, status meetings, and even how the daily standup operates. The tools team only discusses work items that have remained in the queue for an abnormal amount of time, or where the tool developer needs assistance from a peer in its status meetings. This part of the process drove the average length of the team's standup meeting down significantly, from 1.5-3 hours per week on status to about 45 minutes to an hour per week. Expanding this to a team of six developers, the reduced standup time provides 4 to 12 hours more per week to focus on completing the work items in the queue rather than reporting on their status.

Having each team member's work in one easily read location also provided some less tangible benefits. Collaboration has greatly improved among the developers supporting different tooling applications. This drove to more consistency in tooling, work styles, and a better understanding of the organization's strategy and the tools team's place in it.

One example of this is in the performance of our tools supported by different developers. Before Kanban, our test setup application was notoriously slow, taking on the order of 30 minutes to pull all the necessary information down from our central test library. Our metrics reporting tool could pull the same data and even more in approximately one minute. One day during a standup meeting, this discrepancy arose as the tools team was asked to address the test setup application performance. Finally made aware of the size of the discrepancy, the metrics developer assisted in porting his code for tool setup application usage. By the time this performance work was completed, the test setup tooling could pull the required data at the same speed as the metrics application. Collaboration like the example above can certainly occur without Kanban, but the forums and inclusiveness of Kanban created more space for such cross-functional discussions.

Kanban has driven the length of our work tickets to a somewhat consistent size. Larger projects are decomposed into minimum marketable features, and are developed iteratively with rapid prototyping of the defined solution so feedback can be gathered from the involved stakeholders. These techniques enable shorter engagement periods for stakeholders, allowing them to focus on more than tooling design while still providing enough direction for the tool to make a maximum impact.

5 Situations in which to consider using Kanban or not:

5.1 Places where Kanban works well:

- Relative priorities of chunks change often. When priorities are changing faster than items can be pulled off the queue, updating a sorted 1 to N list will be wasted effort.
- Desires of multiple product owners have to be balanced off against each other. In Kanban, stakeholders are told they will meet every week to collectively choose however many items can be started that week (typically a small number such as 3). This reduces the stakeholders' job in any given week to picking the 3 most important items to start that week, rather than engaging in a protracted battle over many weeks to prioritize 100 items according to the stakeholders' varied interests.
- There's no obvious cadence for "sprints" or it's not obvious what a sprint would consist of. Sprints in Scrum perform several purposes – control WIP, provide a cadence for integration and deployment, and provide a cadence for stakeholders so they can change their minds as often as needed. Kanban controls WIP with its explicit WIP limits instead of with sprints. There is a cadence for stakeholders, provided by the weekly stakeholder meetings.
- You are tired of tracking tasks and effort hours. Kanban doesn't require tracking either one. Team velocity is measured in chunks. When initially created, chunks are usually sized into Small, Medium, and Large, and some care is taken that extra-large chunks are broken into smaller chunks, just as epics are broken into stories. Since the team takes on work according to its WIP limit, its average time to process a chunk becomes fairly predictable. Once the average is well known, most stakeholders will use that average in their planning rather than demand an estimate for every chunk.

5.2 Places where Kanban will not work well:

- The work items need to be integrated together *with each other* before being deployed. Kanban doesn't have a good way to handle the grouping. Fixed time boxes or iterations with an integration at the end of each iteration are a better approach.
- The organizational structure is highly cross functional, such that the steps to complete a chunk alternate between teams using Kanban and teams not using Kanban. Two different prioritization methods in use on the same items will cause a lot of delays and frustration.
- Firm commitments must be made well in advance of the delivery dates and last-minute changes in priority are not allowed. The frequent re-prioritization in Kanban would not be necessary, although the rest of the method will probably work.

6 Conclusion

We've found Kanban to be an admirably lightweight method of tightly managing both capacity and throughput of work in an environment of rapid change. It's easier to manage our teams' workloads, see when collaboration would be useful, and avoid working on low-priority items. When we switched from Scrum-style standups to the Kanban-style standups, we found that our standups got shorter, the team members were better able to help each other, and the meeting was more energetic. The work tickets are converging to a consistent size and rapid development of prototypes enable stakeholders to remain engaged in more than just the input queue replenishment.

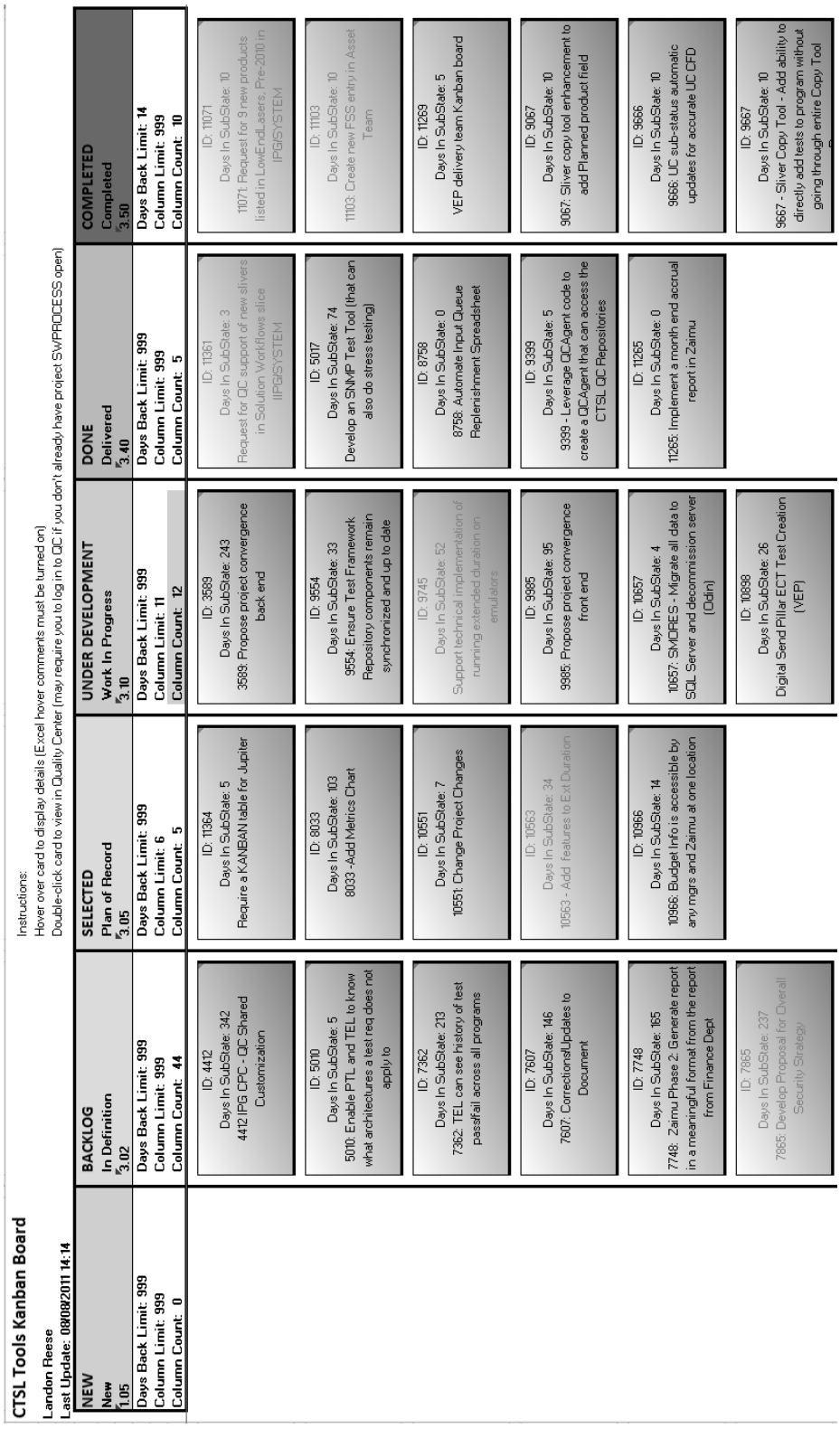
References

[AND2010]: Anderson, David J. *Kanban: Successful Evolutionary Change for Your Technology Business*. Sequim: Blue Hole Press, 2010. Print.

[KNI2009]: Kniberg, Henrik and Skarin, Mattias. *Kanban and Scrum – making the most of both*. InfoQ, 2009. Web. 10 July 2011. <http://www.infoq.com/minibooks/kanban-scrum-minibook>.

[REI2009]: Reinertsen, Donald G. *The Principles of Product Development Flow*: *Second Generation Lean Product Development*, ch. 3. Redondo Beach: Celeritas Publishing, 2009. Print.

7 Appendix A


CTSL Tools Kanban Board
Landon Reese
Last Update: 08/08/2011 14:14
Instructions:
Hover over card to display details (Excel hover comments must be turned on)
Double-click card to view in Quality Center (may require you to log in to QIC if you don't already have project SWPROCESS open)
NEW
New
1.05
Days Back Limit: 999
Column Limit: 999
Column Count: 0
BACKLOG
In Definition
3.02
Days Back Limit: 999
Column Limit: 999
Column Count: 44
ID: 4412
Days In SubState: 342
4412:IPG CPC - QIC Shared Customization
ID: 5010
Days In SubState: 5
5010: Enable PTL and TEL to know what architectures a test req does not apply to
ID: 7362
Days In SubState: 213
7362: TEL can see history of test pass/fail across all programs
ID: 7607
Days In SubState: 146
7607: CorrectionsAUpdates to Document
ID: 7748
Days In SubState: 165
7748: Zaimu Phase 2: Generate report in a meaningful format from the report from Finance Dept
ID: 7865
Days In SubState: 237
7865: Develop Proposal for Overall Security Strategy
SELECTED
Plan of Record
3.05
Days Back Limit: 999
Column Limit: 6
Column Count: 5
ID: 11364
Days In SubState: 5
Require a KANBAN table for Jupiter
ID: 8033
Days In SubState: 103
8033 -Add Metrics Chart
ID: 10551
Days In SubState: 7
10551: Change Project Changes
ID: 10563
Days In SubState: 34
10563 - Add features to Ext Duration
ID: 10966
Days In SubState: 14
10966: Budget info is accessible by any mgrs and Zaimu at one location
UNDER DEVELOPMENT
Work In Progress
3.10
Days Back Limit: 999
Column Limit: 11
Column Count: 12
ID: 3589
Days In SubState: 243
3589: Propose project convergence back end
ID: 9554
Days In SubState: 33
9554: Ensure Test Framework Repository components remain synchronized and up to date
ID: 9745
Days In SubState: 52
Support technical implementation of running extended duration on emulators
ID: 9985
Days In SubState: 95
9985: Propose project convergence front end
ID: 10657
Days In SubState: 4
10657: SMCIRES - Migrate all data to SQL Server and decommission server (Odin)
ID: 10898
Days In SubState: 26
Digital Send Pillar ECT Test Creation (VEP)
DONE
Delivered
3.40
Days Back Limit: 999
Column Limit: 999
Column Count: 5
ID: 11361
Days In SubState: 3
Request for QIC support of new slivers in Solution WorkFlows slice IIPG/SYSTEM
ID: 5017
Days In SubState: 74
Develop an SNMP Test Tool (that can also do stress testing)
ID: 8758
Days In SubState: 0
8758: Automate Input Queue Replenishment Spreadsheet
ID: 9399
Days In SubState: 5
9399 - Leverage QICAgent code to create a QICAgent that can access the CTSL QIC Repositories
ID: 11265
Days In SubState: 0
11265: Implement a month end accrual report in Zaimu
COMPLETED
Completed
3.50
Days Back Limit: 14
Column Limit: 999
Column Count: 10
ID: 11071
Days In SubState: 10
11071: Request for 9 new products listed in LowEndLasers. Pre-2010 in IPG/SYSTEM
ID: 11103
Days In SubState: 10
11103: Create new FSS entry in Asset Team
ID: 11269
Days In SubState: 5
VEP delivery team Kanban board
ID: 9067
Days In SubState: 10
9067: Sliver copy tool enhancement to add Planned product field
ID: 9666
Days In SubState: 10
9666: UC sub-status automatic updates for accurate UC CFD
ID: 9667
Days In SubState: 10
9667 - Sliver Copy Tool - Add ability to directly add tests to program without going through entire Copy Tool

Audit Effectiveness - *Assuring Customer Satisfaction*

By Diane Clegg, Simon Lang and Jeff Fiebrich
Diane.Clegg@Freescale.com, Slang@Freescale.com, J.Fiebrich@Freescale.com
Freescale Semiconductor, Austin, Texas 78602

ABSTRACT

Whether a company is large or small, audits are important factors in continually improving business practices. Ultimately, improved business practices will assure customer satisfaction and drive project effectiveness. Audits can be used to identify best practices to be shared across the company as well as identify areas needing improvement. Good planning is imperative to a successful audit. Audit planning requires not only creating a schedule, but ensuring the appropriate people are available to be audited. Preparation is the key element in planning an effective and efficient audit. This will include scheduling, recruiting the team, training and kick-off meetings. Then determine the right skill set you will need to fill the auditor role. This role will be critical to preparing auditees, proper reporting, avoiding audit challenges and, if necessary, managing 3rd party audits. No matter what combination of audits is right for your organization, planning and monitoring of the system is key. Drive for an understanding of the value your overall audit program brings to your organization and your customers. Align your Key Process Indicators with your company's strategies, track them, change things when necessary and, above all, maintain the integrity of your audit program.

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 is the co-author of the book, 'Collaborative Process Improvement', Wiley-IEEE Press, 2007.*

Simon Lang *is a Software Quality Manager for Freescale Semiconductor. Simon has 10+ years of experience in the High-Tech industry. He has most recently lead various improvement efforts for one of the software organizations using CMMI methodologies and Lean tools. Simon has degrees in business management and is a certified internal auditor, Lean facilitator as well as a Six Sigma green belt.*

Diane Clegg *is a Quality Systems Engineer for Freescale Semiconductor, Inc. She is an America Society of Quality certified Internal Quality Management System (QMS) Auditor. With fifteen years of experience in the semiconductor industry, Diane served as the Project Manager for the Freescale Document Management System (DMS) project implemented via SAP.*

1. Introduction

Whether a company is large or small, audits continually improve business practices. Audits can be used to identify best practices to be shared across the company as well as identify areas needing improvement. Everyone should know by now that audits are performed ultimately to assure the customer is getting what they want, when they want it, and with an acceptable quality.

Your initial audit will always provide you with a baseline upon which you can improve going forward. One of the most critical aspects to achieve continuous improvement and get the maximum benefit out of your audits is to document what you do and do what you document. It is difficult to understand a process that you can't fully describe and even harder to improve it. Projects usually are picked to be subject to external audits often get special preparation and do not necessarily reflect every single project in the organization. All processes contribute to the final result, however, so management needs to be aware of the various types of audits available to them and what goal(s) they want to achieve through them.

Ideally, you have different layers of audits within an organization to achieve maximum impact and results. Group internal, peer-review type audits can be very powerful to use in an organization's day-to-day activities. Examples of these reviews include design reviews or software code reviews. Although relatively resource intensive, errors found during these activities should be recorded. They provide important information for various processes, including measuring defect phase containment effectiveness. Another type of audit can be the phase gate audit, which applies to both waterfall and more iterative project development methodologies. Essentially, this reflects back on the consistent institutionalization of your organization's documented processes (or lack thereof). Feedback gathered during phase gate audits is essential and, in turn, provides a baseline for future improvements.

Finally, audits by an independent Quality organization whether it be company internal or an external ISO certification body can be great motivators and rewarding for internal teams. This type of audit typically gets the best organizational exposure and management attention. Depending on the goal of these audits, they are a great way to show the level of commitment to quality and continuous improvement to your internal and external stakeholders – your customers.

2. Audit Planning

Good planning is imperative to a successful audit. Audit planning requires not only creating a schedule, but ensuring the appropriate people are available to be audited. Preparation is the key element in planning an effective and efficient audit.

Planning for an audit begins with selecting the business process(es) to be audited. This could be determined by an external standard to which the company is required to comply or processes deemed by management needing improvement.

Once the processes have been selected, the appropriate auditors and auditees are determined. Typically, both would be determined by someone having expertise in the area being audited (e.g., someone with a software background will be better at auditing software processes than an auditor without such expertise). Logistics include timing / availability of the auditee and location for the audit itself. Time management is important to ensure that the audit is effective and efficient.

Once the auditors and auditees have been determined and logistics completed, a kick-off meeting is held to introduce the group to the audit process as well as set expectations. This kick-off meeting with outline

the purpose of the audit, the areas to be audited, the project(s) involved and an explanation of what will be done with the results. The team will also be made aware of the potential impact that business processes have on the customer.

3. Auditor's Role

Process auditing starts by looking at how the processes function and how they link elements. Many processes are cross-functional or involve several departments. The auditor should use Plan Do Check Act (PDCA) to identify the parts of the standard necessary for the function of each process and should build the audit around open-ended questions, which reveal more about each process.

The interviews should start with the process owner - the person who can provide the best information about the operation of the process. Other key personnel should be interviewed to get a complete picture. If questions arise, start the discussion with the appropriate clauses of the standard. Always refer to the exact wording, and clarify if necessary.

The auditor should obtain objective evidence by performing a walkthrough of steps of each process. When a finding is identified, the auditor should sit down with the process owner and other implementers to describe the finding in relationship to the wording of the standard.

It is the auditors' responsibility to ensure that the auditee is well-trained and prepared. This is especially true for external or third-party audits. Coaching by the auditor improves upon the efficiency of the audit and is a good learning tool for all. This coaching enables the auditor to better understand the process for which they'll be auditing, and it enables the auditee to understand the requirements to which they must adhere. Be careful, however, not to "over coach" – an auditee should feel comfortable giving an honest accounting of the processes as they understand how they are intended to work, not a recitation of what they think the auditor is looking for.

External audits require a signed contract between the external party performing the audit (e.g., Lloyd's Register Quality Assurance – LRQA) and the company being audited. These audits incur a cost of which selected auditors may be required to manage. Invoice tracking is an important role to ensure that payments are being made in compliance with the contract.

4. Verification Audits

Verification audits are part of what Joseph Juran called the "little q" (quality control, tactical tools) as opposed to the "big Q" (quality assurance or managements systems). System thinking is important, but you can't lose sight of the everyday tools necessary to ensure processes are controlled and risks are minimized.

Supply chain management, outsourcing, process or product complexity and sophistication, certified suppliers and operators, global economies and risk of field failures have all increased the need for ongoing verification. In addition, verifications audits need to be performed when there are routine changes in suppliers, equipment, process settings, methods, requirements or personnel. (J.P. Russell, November 2009)

You need to understand the process and product or service you are going to audit. Reviewing the procedure, specifications and records is a good starting point. If there is no procedure, you may need to ask the auditee to provide a description of the processes. Talk to people to get the information you need.

There are several tools available that can help you understand the process. They include:

- ◊ Process flow diagrams, flowcharts or process mapping
- ◊ Cause and effect, turtle and tree diagrams
- ◊ Process Effectiveness Reports
- ◊ Failure mode effects analysis
- ◊ Training documents
- ◊ Inspection checklist
- ◊ Procedures
- ◊ Bill of materials, quantities and specifications

Find out about process or product history, including:

- ◊ Nonconformance reports and trend analysis
- ◊ Internal and field failures
- ◊ Corrective actions
- ◊ Process or product changes, date and nature of changes
- ◊ Operator or technician changes
- ◊ Revalidation history
- ◊ Customer complaints

Verification audits are part of a robust risk-management process to mitigate potential unacceptable losses. The frequency of verification audits depends on degree of risk and performance history.

Processes and the environments in which they take place are constantly changing. Verification audits are key tools to ensure sustainability of the management system. Results come from checking, not expecting. (J.P. Russel, 2003)

5. Third-Party Audits

There is a subtle dance organizations choreograph and perform in the weeks prior to a scheduled third-party audit. Besides ensuring the organization puts its best foot forward, many of the machinations are intended to ensure failure is avoided.

But, if we stop and contemplate why we think of audit results as pass or fail, we may become more aware of how our mentality affects the way we behave during an audit and how that mind-set influences others.

Part of preparing for the third-party auditor is blitzing the organization with mini-audits. These unscheduled walkthroughs of office and manufacturing areas with eyes peeled are intended to identify anything out of the ordinary, such as untagged nonconforming material, unlabeled or past-due inspection and measuring equipment, general housekeeping, obsolete or uncontrolled documents, and outdated communication boards with performance data.

As soon as a discrepancy is observed, it is brought to the attention of someone with the appropriate level of authority in the area targeted for correction. Then, additional blitzes are performed with a heightened sense of urgency and mission, focusing on the observed problem.

Naturally, all identified discrepancies are fixed right away. There is a sense that things just got a little bit better, and we become a little less vulnerable to the auditor's sword . . er, pen.

The obvious problem is that the system did not get better. The discrepancies observed were symptomatic of deficiencies in the systems' processes that continue to lurk beneath the veneer of calm assuredness that all is well. This only promises the problems will return unless causes are identified and effectively resolved with robust corrective actions.

6. Appreciative Audits

Appreciative inquiry is a discovery method that includes "the art and practice of asking questions that strengthen a system's capacity to apprehend, anticipate, and heighten positive potential."

An appreciative audit helps reveal and enhance what's correct or discard what's not. This creates a value-added experience that encourages the workforce by building solutions based on the fundamental truth that in every company, department or project, something works.

The name, appreciative audit, stems from a hybrid of "internal audit" and " appreciative inquiry." The appreciative internal quality audit has some fundamental differences from traditional internal quality audits. It evolved through a process of collaboration where some of the following critical key questions were answered:

- ◊ What if revealing system effectiveness and best practices was just as likely as finding non-conformance?
- ◊ What if the act of asking audit questions began a process of change for the better?
- ◊ What if audit results became a key input to a CEO's strategic plan?
- ◊ What if audits become something to look forward to?
- ◊ What if the events outlined in the previous questions could be achieved without sacrificing conformance to international and industry standards?

Appreciative audits are an ideal internal audit alternative for organizations that would like to carry forward what they are doing best and encourage improvement in the areas they want to do better. Appreciative audits introduce an innovative way to revive a tired internal audit process and jumpstart the improvement cycle, without sacrificing conformance to required standards. (Jon Morris, October 2008)

7. Strategic Alignment

One potential benefit of a mature audit program is its ability to support the execution of the organization's strategy. In their groundbreaking book *The Balanced Scorecard*, Robert Kaplan and David Norton provided a model for measuring the alignment among an organization's key internal, financial, support and customer-oriented processes and its vision, mission and strategy.

In their later book, *Strategy Maps - Converting Intangible Assets into Tangible Outcomes*, they cite research that indicates 70 to 90% of organizations fail to realize success from their strategies. In most instances, this failure is not due to poor strategies, but rather from failure to properly deploy and execute the strategy.

An internal audit program, coupled with a properly deployed strategic plan and a well-designed system of indicators, can help an organization execute its strategy and achieve its performance goals. This program will include annual audits, quality management system audits and auditing for strategic alignment. Annual

audits typically focus on very high-level processes (macro-processes) and systems, while quality management system audits typically focus on process and associated sub-processes. Annual audits provide an overall indicator of the performance and gaps in the organizations. Quality management system audits drill down to the processes that drive overall performance, providing specific information needed to close the gaps noted in an annual audit.

Unfortunately, auditing for strategic alignment is not always easy to do. First, the strategic elements must already be in place. These elements include a properly formulated strategic plan, a robust scorecard of strategic indicators, and strategy maps or a process architecture that link the strategic goals and initiatives to the internal processes that support them.

In addition, the organization must have senior managers who recognize the potential of the audit program as a deployment aid and support audits of strategic alignment and performance. Finally, the organization must also have experienced auditors with the knowledge, perspective and credibility needed to successfully evaluate and report on areas of strategic alignment (or misalignment) within the organization's systems and processes.

Internal auditing for strategic alignment and execution requires that the auditor understand the organization's desired strategy and how each process fits into this strategy.

It is fair to say that not all processes are created equal, especially when considering their contributions to the success or failure of the strategic plan. Auditors must be shown or be able to develop the connections between the strategy and the processes being evaluated. Auditors will then be in a position to determine the degree of alignment between the strategy and the process inputs, resources, metrics and performance in support of the strategic goals. To accomplish this, internal auditors should be provided additional training that includes:

- ◊ The organization's vision, mission and values.
- ◊ The organization's overall competitive strategy (for example, low-cost provider, customer intimacy or product leader).
- ◊ The current strategic plan and initiatives needed to execute the chosen strategy.
- ◊ The balanced scorecard or other dashboard of high-level strategic indicators used by the organization to measure strategic performance.
- ◊ The organization's system for establishing process metrics and reporting process performance.

These topics help provide the business perspective the auditors will need when evaluating what is most important and where they should allocate their time during the evaluation.

In addition, auditors should receive training in tracing the links from the high-level strategic indicators to the lower-level business processes that drive them. This training and experience can be provided in-house.

8. Audit Program Value

Properly directed, internal audit program resources can help an organization stay focused and uncover new improvement opportunities in the context of the company’s overall strategy and focus on the customer and customer satisfaction.

Unfortunately, the effectiveness of many audit programs is limited to counting findings or other simplistic measures. Effective audit programs, however, should provide insight and support the organization's

objectives. For some organizations this can be a challenge since they are not sure how to gauge the effectiveness of their audit program.

At first glance, gauging the effectiveness of your internal audit program may seem easy. If you have accomplished your objectives, the audit program is effective. But as you start to list the organization, department and audit program's objectives, things start to get fuzzy. What methods will you use, and what measures do you need to monitor? How will you evaluate the impact to your customers?

After a few minutes of scratching you head, you may be tempted to go back to counting audits conducted, nonconformities issued and closed corrective actions as your key performance indicators (KPIs) - but please resist.

Effectiveness of the audit program should be of interest to audit managers and auditor team members; managers, because they should be making the right decisions to continually improve the internal audit function and auditors because they will be asked for their input and agreement on the performance parameters by which they will be judged.

As you can see, the number of audits conducted or corrective actions requests closed are not adequate KPIs of the effectiveness of the internal audit program. Counting audits addresses only the process and not the outcome (product).

Audit program performance indicators should be based on objectives that reflect the audit program mission and organizational objectives and goals. The organizational objectives and goals define the big picture of where you want to be one to five years from now and should be well-understood and prioritized.

Not all objectives are equally important to the organization or audit program. Objectives and goals may be lumped into three groupings:

- ◊ Group 1 - critical for the organization to operate. Top management wants to know if the organization adheres to all applicable standards to ensure critical licenses and certifications will be retained.
- ◊ Group 2 - Necessary for day-to-day management.
- ◊ Group 3 - Required for advancement and growth.

The group numbers are not necessarily the order of importance of the individual organization's objective or goal.

Once you know the objectives, you, as the manager, or your management team can develop strategies to achieve the audit program objectives. The strategies will be based on the type of organization, the organizational culture and resources. Some of the strategies may be simply to formalize what you are already doing.

An objective may be to maintain continuous compliance using fewer resources, with audit program strategies or tactics that could include establishing a network of audit advisors for areas needing assistance to comply based on past results or establishing and implementing an e-audit program.

9. Summary

By applying intelligence to the audit process, which will include building in the audit process to the overall project and process planning activities can result in big gains for the effectiveness of internal processes. This effectiveness will, in turn, display itself in bottom-line results because customer satisfaction (be they internal, or external) will be increased.

To do this, remember to begin your audit management by

- Planning it - this will include scheduling, recruiting the team, training and kick-off meeting management.
- Determine the right skill set you will need to fill the auditor role. This role will be critical to preparing auditees, proper reporting, avoiding audit challenges and, if necessary, managing 3rd party audits.

Every process should have effectiveness measures and the audit process should be no exception. Be sure to understand how your audits are strategically aligned to the business and figure out what you need to know about non-conformities and the overall audit program that will allow you track what is important.

Consider also the types of audits you will conduct and ensure your selected auditor is the right person for the job.

- Verification audits to ensure quality control and tactical practices will require that the auditor understands the product or service that is being audited
- Third-party audits will need to be choreographed carefully prior to the actual audit event
- Appreciative inquiry will require an understanding of how to manage a less-traditional, more creative type of internal audit.

Drive for an understanding of the value your overall audit program brings to your organization and your customers.

Then tell everyone about it. Pass it on, talk it up and make it sing. Perception is real – help ensure that audits are perceived as essential to quality management systems that *work*.

10. References & Permissions

Standards and Deviations: The Role of Routine in Testing

Michael Bolton, Develop Sense, Inc.

Eliminating variation makes sense in manufacturing, where the goal is to make zillions of compatible widgets based on the same pattern. Yet software development isn't much like manufacturing; it's more like design. Which activities of design can be standardized? Which ones can't? What aspects of software development are based on explicit or explicable knowledge? What knowledge do we need in order to apply a standard successfully? What can we learn from other disciplines like art and music? What parts of architecture are standardized, and what parts are not? How can we have "industry standards" without a clear notion of which industry we're talking about?

It's harmful variation we want to eliminate, but systematic observational errors and compulsive standardization can make it simple to throw out the baby with the bathwater. In many places where people claim we need standards, at most we need guidance. In many places where people claim we need scripts, at most we need checklists. When we reduce deviation, we reduce opportunities for exploration, discovery, investigation, and *positive* deviation from overly simplified norms.

Michael Bolton has been teaching software testing on five continents for ten years. He is the co-author (with senior author James Bach) of Rapid Software Testing, a course that presents a methodology and mindset for testing software expertly in uncertain conditions and under extreme time pressure. He has been Program Chair for the Toronto Association of System and Software Quality, and Conference Chair (in 2008) for the Association of Software Testing. He wrote a column in Better Software Magazine for four years, and very sporadically produces his own newsletter.

Michael lives in Toronto, Canada, with his wife and two children. He can be reached at mb@developsense.com, or through his Web site, www.developsense.com.

Standards and Deviations: The Role of Routine in Testing

Michael Bolton
DevelopSense
http://www.developsense.com

PNSQC 2011

Updates

- This presentation is ALWAYS under construction
- Updated slides at http://www.developsense.com/past.html
- All material comes with lifetime free technical support

What Problems Do Standards Purport to Address?

- The Knowledge Problem
- The Skills Problem
- The Credibility Problem
- The Management Problem
- The Deviance Problem

Introducing Standards!

- Common and repeated use of rules, conditions, guidelines or characteristics for products or related processes and production methods, and related management systems practices.

note: non-standard spelling

- Source: http://www.standards.gov (NIST)
- Standardization: "the process of developing and implementing technical standards"
- "Standardization is defined as best technical application consentual wisdom inclusive of processes for selection in making appropriate choices for ratification coupled with consistent decisions for maintaining obtained standards."
  - Wikipedia, "Standardization"

Regulations

- "Governmental regulations, also called rules, specify mandatory (legal) requirements that (1) must be met under specific laws and (2) implement general agency objectives."
- "An agency may adopt a voluntary standard without change by incorporating the standard in an agency's regulation or by listing (or referencing) the standard by title. For example, the Occupational Safety and Health Administration (OSHA) adopted the National Electrical Code (NEC) by incorporating it into its regulations by reference."
  - http://www.standards.gov (NIST)

McLuhan's Laws of Media

- Standards are *media*
- McLuhan proposed that every medium
  - *extends* some human capability
  - *retrieves* the idea of some currently obsolescent medium
  - *obsolesces* some existing medium
  - when overheated, *reverses* into the opposite of its original or intended effect
- A medium, by mediating, reduces immediacy

Laws of Media Tetrad Example: Smartphone

Extends…
- Presence
- Communication
- Availability
- Organization
- Memory

Retrieves…
- Transporter
- Telephone
- Magic carpet
- Secretary
- Library

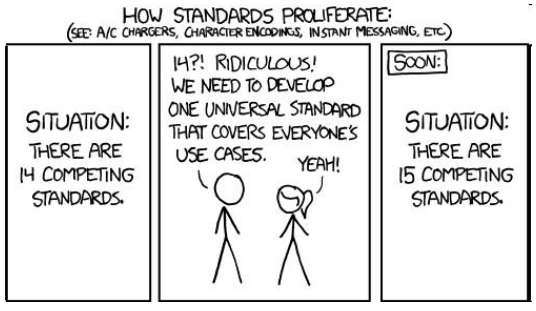
Obsolesces…
- Office
- Pager
- Meetings
- Appointment book
- Diary

Reverses into…
- Absence
- Silence
- Inaccessibility
- Chaos
- Forgetfulness

ISO/IEC 29119: The Goals

- The aim of ISO/IEC 29119 Software Testing is to provide one definitive standard that captures vocabulary, processes, documentation and techniques for the entire software testing lifecycle.
- From organisational test strategies and test policies, project and phase test strategies and plans, to test case analysis, design, execution, reporting and beyond, this standard will support testing on any software development or maintenance project.

XKCD on Standards

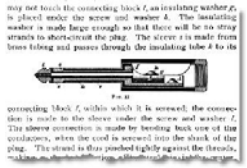


http://xkcd.com/927/

Six Assumptions For Testing Standards

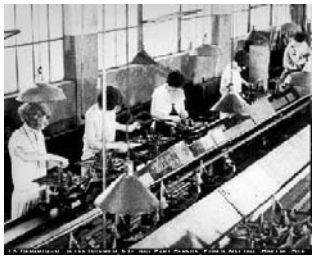
1. We, the standard's authors, are the right people to make them.
2. We've made the important stuff explicit.
3. Our knowledge is complete.
4. There's no controversy
5. Standards won't get in the way.
6. There will be no coercion.

Martian Headsets



Joel Spolsky, http://www.joelonsoftware.com/items/2008/03/17.html

Software Development Is Not Much Like Manufacturing

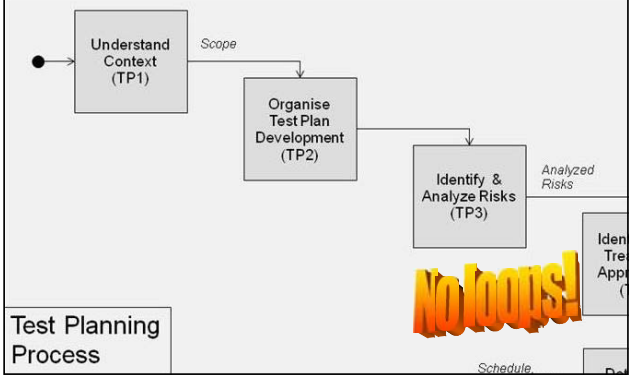


- In manufacturing, the goal is to make zillions of widgets *all the same*.
- Repetitive checking makes sense for manufacturing, but…
- In software, creating zillions of identical copies is not the big issue.
- If there is a large-scale production parallel, it's with *design*.

Software Development Is More Like Design

- If existing products sufficed, we wouldn't create a new one, thus…
- Each new software product is novel to some degree, and that means a new set of relationships and designs every time.
- Should the process of design be standardized?

Testing As Assembly Line



Problems With Testing Standards

- Misperceptions in the institutionalization of knowledge and behaviour
- Oblivion to the role of tacit knowledge
- The ontology problem
- Ignorance of context
- Delegation of authority and expertise
- Reification
- Suppression of diversity and The Fundamental Regulator Paradox

Problems With Testing Standards

- Evolution of (bad) standards into regulation
- The time-binding problem
- Market manipulation
- The Bogus Maturity Argument
- The Bogus Research Problem
- Goal displacement

Deviation and Science

Alexander Fleming

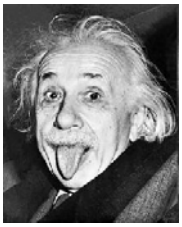
Penzias & Wilson

Henri Becquerel

Galileo

Charles Goodyear

Science *Requires* Deviation



So You Want Process Improvement?

The Positive Deviance approach is
- an asset-based,
- problem-solving, and
- community-driven
- approach that
- enables the community
- to discover these successful behaviors and strategies and
- develop a plan of action
- to promote their adoption by all concerned.

Source: The Positive Deviance Initiative
http://www.positivedeviance.org/

Positive Deviance

Positive Deviance is based on the observation that
- in every community
- there are certain individuals or groups
- whose uncommon behaviors and strategies
- enable them to find better solutions to problems than their peers,
- while having access to the same resources and
- facing similar or worse challenges.

Source: The Positive Deviance Initiative
http://www.positivedeviance.org/

Positive? Deviant?

- Positive
  - "doing things right"
- Deviant
  - "engaging in behaviour that others do not"

A tester is someone who knows that things can be different.
— Jerry Weinberg

PD isn't limited to testers, of course.
Anyone, everyone, can contribute.
Testers, as the antennae of the project, should be on the lookout for PD opportunities.

An Example of Positive Deviance

- Problem: Hospital staff wear disposable gowns that can become contaminated by contact with MRSA patients. The garbage gets full and overflows quickly, risking more contamination.

Enter Jasper Palmer

Source: The Positive Deviance Initiative
http://www.positivedeviance.org/

Testing Is Strengthened By Diversity

- Educational experience
- Writing skill
- Cultural background
- Domain knowledge
- Temperament
- Gender
- Programming skill
- Testing experience
- Age
- Experience in the current culture
- Experience *outside of* the current culture

What Is Testing?

Software testing is the investigation of *systems* composed of people, computer programs, and related products and services.

- Excellent testing is not a branch of computer science
  - focus only on program code and functions, and you leave out questions of *value* and other relationships that include people
- To me, excellent testing is more like *anthropology*
  - highly multidisciplinary
  - doesn't look at a single part of the system
- Anthropologists investigate many things
  - biology (human mechanisms; human "code" and "hardware")
  - archaeology (human history)
  - linguistics (human communication)
  - cultures (what it means to be human)

Are Standards The Only Way To Advance? What Else Could We Use?

- Guidance
- Checklists
- Guideword Heuristics
- Diverse half-measures
- Humility
- Uncertainty
- Practice
- Craftsmanship

Are Standards The Only Way To Advance? What Else Could We Use?

- Testing for adaptability
- Personal syllabi
- Serious study of measurement
- Coaching, mentoring, apprenticeship
- History
- Philosophy
- Continuous conversation
- Extending increasing authority

Conclusions

- Standards can be useful
- Standards can be profitable
  - but mostly for the standards-makers
- Instead of standards, what we really need in testing is skill.

Updates

- This presentation is ALWAYS under construction
- Updated slides at http://www.developsense.com/past.html
- All material comes with lifetime free technical support

Book References

- The Shape of Actions
- Tacit and Explicit Knowledge
  - Harry Collins
- Seeing Like a State: Why Certain Schemes to Improve the Human Condition Have Failed
  - James C. Scott
- Ideas on the Nature of Science
  - David Cayley
- The Power of Positive Deviance

Book References

- The Checklist Manifesto
  - Atul Gawande
- The Black Swan
- Fooled by Randomness
  - Nassim Nicholas Taleb
- Secrets of a Buccaneer Scholar
  - James Bach
- Sciences of the Artificial
  - Herbert Simon
- Validity and Reliability in Qualitative Research
  - Kirk & Miller

Book References

- Blink
- Outliers
  - Malcolm Gladwell
- Tools of Critical Thinking
  - David Levy
- The Visual Display of Quantitative Information
- Envisioning Information
- Visual Explanations
- Beautiful Evidence
  - Edward Tufte

References: Cem Kaner

- The Ongoing Revolution in Software Testing
  - http://www.kaner.com/pdfs/TheOngoingRevolution.pdf
- Software Testing as a Social Science
  - http://www.kaner.com/pdfs/KanerSocialScienceSTEP.pdf
- Software Engineering Metrics: What Do They Measure and How Do We Know? (with Walter P. Bond)
  - www.kaner.com/pdfs/metrics2004.pdf
- Approaches to Test Automation
  - http://www.kaner.com/pdfs/kanerRIM2009.pdf
- Lessons Learned in Software Testing
  - Kaner, Bach, & Pettichord

References: Jerry Weinberg

- Perfect Software and Other Illusions About Testing
- Quality Software Management
  - Volume 1: Systems Thinking
  - Volume 2: First Order Measurement
- Quality Software Management: Requirements Before Design
- An Introduction to General Systems Thinking
- The Psychology of Computer Programming

Personal Kanban: Visualizing, Understanding, and Communicating Your Work Load

Jim Benson, Modus Cooperandi Inc.

If we can't see our work and understand it, how do we expect others to value it? Testing is often undervalued by project managers and development teams, but quality products rely upon it. Often, this is because testing's day- to- day actions and its unpredictable work load are invisible to others. Jim Benson will show how visualization of work by testers can greatly improve relations and create a less stressful workload.

Jim discusses Personal Kanban in this part tutorial / part social psych lesson / part QA/QC case study extravaganza. Jim also thinks Powerpoint is boring, uses no slides, and prefers instead to actually talk to people.

Jim Benson is the CEO of Modus Cooperandi Inc, a collaborative management consultancy that specializes in creating cultures of continuous improvement for knowledge workers. With 25 years of experience in very large project management (building freeways, light rail systems, software, and international development projects), Jim has come to the realization that projects veer off-track because of lack of information and overly aggressive planning. Since starting Modus Cooperandi back in 2007, Jim has worked with the United Nations, the World Bank, NBC Universal, British Telecom, The Library Corporation and others to get individuals, teams, and the organizations they work with collaborating and exchanging information effectively. To this end, he created the Personal Kanban methodology, and recently co-authored the book Personal Kanban: Mapping Work | Navigating Life with Tonianne DeMaria Barry.

The Ladder of Unmanaged Conflict

Jean Richardson

jean@azuregate.net

Abstract

Because building software is a social process, navigating conflict in the software development process is a large part of what occupies team member time. It seems to be perennially scary to deal with conflict. NOT dealing with conflict has a very high cost that is often unacknowledged; it can destroy teams and entire organizations. Using proven research and anecdotes drawn from the speaker's experience, this paper will address:

- A model to help us think about various ways of responding to conflict
- How conflict escalates and the signs of escalation
- What effective conflict engagement actually is
- Basic techniques for de-escalating conflict
- Reasons not to fear dealing with conflict

Attendees will take away:

- A simple model for identifying conflict processes.
- Techniques for short-circuiting escalation.
- A fresh perspective on the "scariness" of conflict.

Biography

A software development professional since 1989, Jean Richardson is experienced in adaptive and predictive project management, writing, training, public speaking, and requirements and business analysis. She holds a B.A. in English with minors in economics and administrative systems management and is completing an M.A. in Organizational Communication in Winter 2011. Her master's thesis is titled In Your Own Hands: Personal Integrity and the Individual's Experience of Work Life *and focuses on personal integrity through the lens of agile methods, specifically, Scrum. She holds PMP, CSM, CSPO, and ITIL certifications, has met the Oregon Department of Justice standard for court-based mediators and has been mediating in the court system for over 10 years.*

As a consultant, her client list boasts a wide range of businesses including ADP, Chrome Systems, CoreLogic, Intel, Freightliner, Kaiser Permanente, Kryptiq Corporation, Mentor Graphics, Oregon Health Authority, The Regence Group, Tripwire, and US Bank. EPHT GEORGE, a project Jean managed for the Public Health Division of the Oregon State Department of Human Services, won the Project Management Institute 2009 Project of the Year Award for the Portland Chapter.

1. Introduction

In one company, a group of leaders sit around a table concerned that their division is going to miss its annual goals again. This is not the first time this has happened. This is the third year in a row. The three years prior to that, the division had made its goals—technically, but it had squeaked by, and those who were around back then had to make a strong case to get executive management to agree that the goals had been met. Some of them are middle managers responsible for functional areas. Some of them are leaders of the larger teams in the division. As they talk about the situation they find themselves in at the beginning of the third quarter, they are interrupted several times by messages being brought in by the administrative assistant for the director who is in the room. A serious argument has broken out among team members in a planning session for the upcoming release. The message sent back out of the room to those team members is that they are "all adults," and they should resolve the conflict for themselves. The managers are focused on important strategic planning for the remainder of the year . . .

In another company, a senior contributor who has been interested in moving into leadership was appointed to a lead position on a project critical to the success of the organization several weeks ago. However, the characteristics that previously caused him to develop a reputation for raging and factionalizing the work group have begun to re-emerge under the pressure of transitioning to leadership. The leadership team that placed him in the role is hoping that someone else will take care of the problem; most of them have had unpleasant encounters with him and are tired of dealing with his human resources complaints about them. They realize that he could have a deleterious effect on the team he has been assigned to lead, but they're hoping for the best. They are discussing an incident in which two of them saw him yelling at a female contractor last week. Another one noted that she heard he took his team out for a two hour lunch the previous Friday, though she also knows the team is way behind on their commitments. They all look at each other. Is anyone going to take this on?

In over 20 years of experience in the software industry I have seen many teams wrestle with "teaming," the modus operandi of working together toward common goals with people you may or may not like or choose to spend time with outside of the workplace. In many environments, people are not even comfortable enough with the term "conflict" to use it, and so they resort to euphemisms like "problems," "problem solving," "challenges," "opportunities," and so on. To set the context for our conversation today, we will define the term "conflict" as:

> That which occurs whenever two or more people perceive they have opposing interests.

This is not a "personality conflict" and this paper does not address the topic of personality conflicts. That's a different topic, though it interacts with the topic of this paper. The focus of this paper is to understand how conflict typically evolves if it is neither managed nor resolved and to learn a simple technique to prevent an escalation of conflict and, instead, move toward resolution.

2. Responding to Conflict

To begin, it's important to understand how to identify an interest as well as how to identify the close cousins it travels with, issues and positions. When conflict arises it can be identified in a number of ways which we are all familiar with: raised voices, scowling looks, phrases such as "I disagree," "You're not hearing me," and so on. But it can also be identified by averted eyes, silences, and puzzled expressions. As the conflict evolves someone may well start "defining the problem space" or stating her "position."

During the discussion, you may well hear someone try to calm and organize the discussion by listing issues. From a conflict navigation perspective, these critical terms are defined below:

- **Interests** are expressed in generalized language with reference to the conflict. They are often not concrete. Interests are the keys to finding workable solutions for each issue: a specific need or desire that a person has and wants to satisfy.
- **Issues** are components of a larger problem.
- **Positions** appear to be black and white or binary expressions. There is usually more than one viable position for every interest.

One effective way to navigate emerging conflict is to listen for the interests, table the positions, and collect the issues to help inform the resolution.

Some people, even people in leadership positions, like to take the approach that if we just ignore the conflict or "give it time," the conflict will go away. Sometimes that works, but generally it doesn't. It's important to understand the risks of avoidance, particularly if you are in a leadership position. Carpenter and Kennedy in *Managing Public Disputes: A Practical Guide for Government, Business, and Citizens' Groups* (Carpenter & Kennedy 2001), have provided a concise representation of what tends to happen when conflict is not managed, or resolved. The bullet points below are adapted from their work and listed in order of least to most severe.

The Ladder of Unmanaged Conflict

- ***The Problem Emerges.*** *Initially, parties may express curiosity or mild concern about an issue. When parties receive an unsatisfactory answer, or are ignored, the spiral moves up.*
- ***Sides Form.*** *The longer the conflict remains unmanaged, the greater likelihood that parties involved will position themselves with one side or the other. As people form opinions, they feel the need to get together with others who have similar views. Sometimes, the media finds the differences between the sides to be fertile ground for news stories. The conflict expands as more people learn about it through "word of mouth" and/or the media.*
- ***Positions Harden.*** *People talk more with others of similar views, and less with people with whom they disagree, even in situations not related to the dispute. People become rigid in their perception of the problem and their opponents.*
- ***Communication Stops.*** *Information is exchanged haphazardly, or not at all, between the parties. Misunderstandings are common, and communication becomes increasingly adversarial. Public discussion can turn into public debates. Listening to each other is unpleasant, and effective communication stops.*
- ***Sides Strengthen Their Positions.*** *Individuals gain a sense of power from being part of a group (side), and become ready to commit resources (financial, as well as personal time) to win the battle. People begin to look outside of the dispute community for support and power. Lawyers, supervisors, or other "middle people" come between the parties and prevent face to face negotiation.*
- ***Perceptions Become Distorted.*** *Parties lose objectivity in their perceptions of the conflict, and of the character and motives of their adversaries. Shades of gray disappear, and only black and white remain.*
- ***Sense of Crisis Emerges.*** *It now seems there is little hope in resolving the original dispute. The parties are now willing to bear high costs (emotional, financial, etc.) that would have seemed unreasonable earlier. The goal becomes progressively to win at any cost.*
- ***Outcomes Vary.*** *Litigation, firing or resigning employees, arbitration from outside parties, and even violence are traditional outcomes.* (Richardson & Burk, 2003)

3. How conflict escalates and the signs of escalation

Conflict can escalate in a moment or over time. Escalation is supported by behaviors such as inappropriate avoidance or confrontation. Lack of effective management of emotions or unwillingness to engage in safe and productive conversations can cause parties to escalate the conflict. Escalation is not always accompanied by raised voices, as the Ladder of Unmanaged Conflict shows. For example, when a sense of crisis emerges, it's just as likely for voices to be hushed with heads together in cubes as it is for anyone to raise his or her voice. It's also not unlikely for people to begin to avoid each other—either through not attending meetings or responding to email or simply not looking each other in the eye during small group discussions.

However the escalation manifests itself, you can be sure that stress levels are elevated and that the amygdala, or animal brain, is more active than those parts of the brain more adept at complex reasoning. Emotions are flowing and fear is generally part of the mix, even if it is masquerading as anger.

4. What effective conflict engagement actually is

Conflict engagement is *effective* in a workplace context when the work is moved forward, the work group's work processes are minimally disturbed, and all relationships are preserved to the greatest degree possible—in some cases, even strengthened. As Diana McLain Smith describes at length in *Divide or Conquer: How Great Teams Turn Conflict into Strength* (Smith 2008), conflict can be valuable in strengthening relationships and building effective teams. In fact, work teams which cannot acknowledge the existence of conflict, let alone lack the skills to navigate it are inherently "stuck" and unproductive, since creativity—generativity, which is essential to our work—requires effective conflict engagement.

This does not mean that sometimes effective conflict engagement does not result in an action as extreme as a termination or involuntary removal from a team. What it does mean is that the conflict is addressed in a manner that results in resolution so that attention can be directed away from an ongoing dispute and back toward the work. We are, after all, in the knowledge creation business, and full attention is required for the successful prosecution of that work.

The most desirable form of conflict resolution occurs directly between the individuals involved in the conflict at the time it arises and is initiated and completed by them. Because of this, the direction from the distracted managers in the first example in the introduction makes sense. However, it's also clear that the team members in conflict were appealing to a higher authority to help manage or resolve the conflict. This may have been because they lacked the skills to resolve it themselves or felt they lacked the authority to do so, in which case management's appropriate function is to get the train back on the rails: Model effective conflict engagement and coach those skills into the team members so that, the next time, they can resolve it among themselves.

In *Controlling the Cost of Conflict* (Slaikeu & Hasson 1998) the authors cite a four option model: Avoidance, Collaboration, Higher Authority, and Unilateral Power Play. Collaboration and Higher Authority provide the best outcomes, with Collaboration providing greater long-term mastery and relationship building for all parties than Higher Authority. The table below describes these four options.

Avoidance	Collaboration	Higher Authority	Unilateral Power Play
• No action to resolve the conflict.	• Individual initiative. • Negotiation by the parties. • Mediation by a third party.	• Referral up line of supervision, or chain of command; internal appeals; formal investigation.	• Physical violence. • Strikes. • Behind-the-scenes maneuvering.

"The 'preferred path' for cost control encourages collaborative options first, with higher authority in a backup role" recommend the authors (Slaikeu & Hasson 1998, p. 22). The authors' data shows that, though Collaboration is the most desired and beneficial model, the models most in use in organizations are the Higher Authority and Unilateral Power Play models. The predictable costs from the Avoidance and Unilateral Power Play models are loss of productivity and legal costs. The next most expensive alternative is the Higher Authority model because more resources (managers) must be brought to bear on the problem. While collaboration deals with the problem at it source, preferably through the use of skills already resident in the parties to the conflict. Where that does not occur, management engagement to deal with the problem currently in evidence and to build in those skills so the parties can navigate the problem on their own is the best alternative. Management avoidance can move the conflict down the ladder of unmanaged conflict discussed above.

Mediators talk about conflict resolution versus conflict management. This is a philosophical difference, and collaborative practices tend to favor resolution because it results in the greatest synergy. What is resolution and how do you know when you have achieved it? Resolution is a feeling state though it may be expressed intellectually. Sometimes a facilitator or mediator may ask something like "Do you feel complete with this," or, "Is this conversation complete for you?" as a means of eliciting whether an individual feels the conflict is resolved. Sometimes one or more parties to the dispute may be startled by a sense of the conflict having "disappeared" as though it somehow just left the room. And, not uncommonly, true resolution of a conflict is accompanied by some level of reconciliation with the other part(ies) to the dispute, and even a deeper sense of relational intimacy may suddenly be sensed. This deeper sense of awareness and empathy that accompanies a resolved conflict generally comes about because of new information that is shared, new perspectives that are gained, and a sense of being heard and accurately perceived by another.

5. Basic techniques for de-escalating conflict

Obviously, if conflict is escalating in a physically or verbally violent direction, physically separating the disputants into different locations is important. If you are the person who is suddenly feeling stronger hostile or defensive feelings, the best choice may be to say you need to take a break. But this is the point at which avoidance can seep in to the conflict, so it's important to take action as soon as possible to arrange for a more safe and productive engagement on the content.

A recent client of mine had an interesting repeating pattern of what I came to call "performance art conflict" on one of their teams. Pretty much every day just like clockwork two team members engaged in verbally aggressive conflict about their work with raised voices and often both standing up at their desks in the bullpen contesting like software gladiators. Managers sitting nearby would have to intervene, calm them down, and manage the conflict, but obviously whatever the underlying conflict was that was feeding

this flame was never resolved or the late afternoon performances would have become a thing of the past. Imagine the effect on the rest of the team witnessing—or being subjected to this performance. A technique that might be applied in such a situation would be to lead the disputants away from the work area, inquire into the effect they may or may not have observed they are having on their teammates, surface the underlying conflict, focus on that, and pursue resolution through an interest-based mediation process with the objective of transforming the dynamic in play.

Mediators talk about content versus process in a conflict. The process is the container that holds the content—what the dispute is about. An example of the *content* of a conflict might be that you implied that my team made a commitment that we believe we did not make. The process can be very helpful in de-escalating a conflict. A typical example of formal interest-based *process* is:

- Make an initial agreement about what the problem (the conflict) is about.

- Each participant lists the reasons resolving the conflict is important to us (our interests) and see where our interests coincide. For instance, we may want to have the respect of our peers and performance art conflict may not be the best way to get that respect, but it could be a great way to be seen as a whiner or a scary person to work with.

- Each participant lists what about the resolution (the issues) is important to us—the components of the problem as we see it. This is a discussion activity during which people may start listing solutions to the problem and may need guidance to help them stay out of that space until they are clearer about the components of the problem may be.

- First one of us then the other of us will say how we think we got to this point, and we will listen to each other in a manner that seeks new information while the other is speaking, which may require the support of a third party present to help us hear each other.

- Finally, based on the data that we have collaboratively gathered about the conflict and what it will take to resolve it, we create an action plan. This plan will be focused on meeting our interests, and addressing the issues, and being vigilant for the emergence of ineffective conflict in the future.

In addition to the typical interest-based conflict resolution model above, it is also important to deal with difficult and distracting emotions. As discussed in section 3 above, as stress levels increase, the portion of the brain adapted for dealing with complex problem solving is less accessible. This is a result of both chemicals in the bloodstream which are generated when our fight-or-flight response is triggered and the fact that blood flow in the body actually moves more toward large muscle groups in our arms and legs and away from the brain when we are in fight-or-flight mode. You provide a service when you recognize this in yourself or another and take action to help disengage the fight-or-flight response: After all, it's not likely to be useful to fight or flee a typical conflict in the workplace.

To short-circuit the fight or flight response, first BREATHE in and out slowly and deeply once, twice, and then again to help calm emerge. Slow, deep breaths calm the body and help return the mind to clarity. Next, check in with yourself to see whether it might be a good idea to take a break, get a drink of water, or take a walk. These simple things help you return to a clearer frame of mind. If you are in a meeting and really feel you absolutely cannot leave, you can still take a minute or two to collect yourself.

If you are witnessing the emergence of the fight-or-flight response in another person, it can be effective to get their attention or simply walk up to about an arm's length from them and practice the calming breaths yourself. If they are making eye contact with you, it's not unlikely that they'll notice your change in breathing and change their own; if they are not, your calming presence still likely helps calm them.

It can be helpful to use what some conflict experts call “sponges” to help absorb some of the energy of the emotion in the room. “Sponges” are empathetic phrases such as:

- “That must have been difficult for you.”
- “It’s clear that was upsetting.”

It can also be helpful to use distraction techniques including joining in and escalating the intensity slightly beyond the level currently modeled by the angry person: use this one with care. (Aikido practitioners may recognize this technique.) Responding in such an unexpected way can help the other person “see themselves” and inspire a quick modulation of their behavior.

People who are overtly expressing anger—raging—can both frighten others and draw energy for their emotional outburst from onlookers. In extreme situations, there can be danger of physical violence. Therefore, encouraging an angry person to come away from an audience will be helpful in de-escalating volatile situations.

And, finally, if appropriate accept responsibility for offense and apologize as needed is also effective in de-escalation. However, apologies without “teeth” do not have a positive effect. A complete apology has three parts: acknowledgement of the specific nature of the offense, expression of regret for the effect of the offense, and a specific plan to prevent it from happening again. Note that apologies are not always appropriate, so defaulting to this technique if the situation does not call for it will not likely help address the level of emotionality.

For instance in the case of our performance artists above, suppose that they were representatives of testing and development functions and the most common argument was about whether a bug was fixed or why bugs tended to be repeatedly reopened. Management might very well need to intervene to stop the ineffective engagement, lead them away from the stage, and gain agreement on what is nurturing the pattern of conflict. It might be time pressure and unclear expectations. The interests might include a desire to be seen as competent professionals within their specialties in the eyes of both their managers and their peers and to be respected for their professional mastery. An issue may be lack of a documented defect handling process; another might be lack of clear requirements. A plan might be to collaborate on a team-level defect handling process even if there is no organizational process and to reject unclear requirements or for QA and development to review requirements together with the source of those requirements before coding them. And, in this particular scenario, there may need to be apologies either to each other or to the team who have been repeatedly distracted by this performance.

6. Conclusion

Conflict often does not go away on its own, but left unaddressed, it generally traverses the ladder of unmanaged conflict, harming relationships, work products, and entire work groups along the way as well as wasting time and money. Like most aspects of human relationships, dealing with conflict is often messy. But methods such as interest-based mediation can help provide a safe container in which to hold a conflict so that it doesn’t spiral out of control.

A great motivator for inspiring courage in dealing with conflict is an understanding of the ladder of unmanaged conflict and the knowledge that conflict left unaddressed is likely to worsen rather than resolve itself. If we do value our working relationships—and these relationships are the means through which knowledge work is conducted—then engaging productively in conflict will likely become an almost everyday occurrence. We have more to fear in avoiding conflict and letting it fester than in taking steps to

acknowledge its existence and inquire into how it might be resolved and our working relationship strengthened. The responsibility for doing this lies first with the parties to the dispute themselves, but if they lack the skills or the will to do so, then managers or other leaders in the organization abandon their stewardship responsibilities if they do not step in because unmanaged conflict wastes resources and can result in tremendous damage to teams and individuals.

References

1. Carpenter, S.L. and Kennedy, W.J.D. (2001). *Managing public disputes: a practical guide for government, business, and citizens' groups,* San Francisco: Jossey-Bass.

2. Richardson, J. & Burk, L. (2003), "When Conflict Spirals Out of Control," newsletter of the Software Association of Oregon.

3. Slaikeu, K. A. & Hasson, R. H. (1998). *Controlling the Costs of Conflict: How to Design a System for Your Organization*. San Francisco: Jossey-Bass.

4. Smith, D. M. (2008). *Divide or Conquer: How Great Teams Turn Conflict into Strength*. New York, NY: Penguin Portfolio.

Learning Software Engineering - Online

Kalman C Toth

kalmanctoth@gmail.com

ABSTRACT

It is apparent that software practitioners seeking advanced education and professional development demand programs that blend with their obligations at work and at home as well as yield immediate benefits. Ideally, software engineers should be able to integrate their learning with how they work. Acculturated by their social and business networks, they would prefer to use tools and interfaces that are similar to what they already use. The learning processes and tools they are expected to use when learning should blend with their day-to-day processes to the extent possible.

Offering classroom courses in the edge hours - evenings and weekends - is an approach that meets the needs of many working software professionals seeking advanced education. However, the time such practitioners can devote to the campus commute to attend a few hours of classes is rapidly shrinking. Learning by way of online methods and tools has therefore become increasingly popular over the last several years. E-learning over the web offers a great deal of flexibility over traditional learning modes. Several variants and hybrids of these models can, and have been, implemented, each offering distinct benefits, but also limitations.

My experiences to date suggest that better, more cost-effective, and more relevant learning can be achieved by hybridizing face-to-face and online learning modes. For example, hybrid learning can increase the depth of engagement through discussion forums, provide access for dispersed learners, and also satisfy the demand for a modicum of face-time for those students who are prepared to attend classes on campus. However, “one size does not appear to fit all” – for example, discussion forums appear to be less effective for programming courses, while demonstration videos walking through coding and testing samples seem to be very effective. And online teleconferencing mechanisms supporting shared visual space and jointly authored documents are very effective for work group activities – much like those experienced on-the-job.

This paper describes my experiences and evolution of various learning models and hybrids that I have leveraged to blend certain critical elements of traditional face-to-face and online learning approaches. A spin-off benefit of e-learning is that classrooms are freed up for other purposes, reducing the demand for costly physical facilities. In this paper I explore several blended learning model variants that I have used to teach software engineering courses. The observed benefits and limitations of these learning models and support tools are highlighted; and several outstanding questions and issues for further consideration are raised.

BIOGRAPHY

Kal Toth holds a Ph.D. from Carleton University, Ottawa, Canada and is a registered professional engineer in British Columbia with a software engineering designation. He has worked for a range of companies and universities including Hughes Aircraft, CGI Group, Intellitech, Datalink Systems Corp., Portland State University, Oregon State University, and the Technical University of British Columbia. His areas of specialization and R&D interest include software engineering, project management, information security, and identify management.

1. INTRODUCTION

Portland State's Oregon Master of Software Engineering (OMSE) program has offered a 16-course masters degree to software professionals in Metro Portland (Oregon) since 1998. In 2008, OMSE began to offer a 5-course graduate certificate in software engineering. Courses are also offered on a professional development basis. Over a period of several years, OMSE introduced e-learning and hybrid course offerings to satisfy the growing demand for more learning flexibility. During the development and evolution of these offerings, several variants of these learning models have emerged.

With the proliferation of the web and e-collaboration tools, the distinctions between various learning models have been narrowing. A traditional learning model achieves learning almost entirely through face-to-face sessions in the classroom (lectures and Q&A sessions). An online (only) learning model achieves most if not all learning through online web-based mechanisms. Various "blends" or hybrids of face-to-face and online learning can be designed.

Hybrid learning models combine traditional face-to-face and e-learning mechanisms in various ways. They will generally include some traditional face-to-face instruction in a classroom setting and deliver a substantial portion of the learning content as well as engagement with the instructor(s) and other students by means of Internet-enabled collaboration mechanisms.

Another dimension of hybrid learning has to do with the collaboration processes and tools used, namely, whether instructors and learners communicate asynchronously ("time-shifted") and/or synchronously ("at the same time"). Asynchronous collaboration is achieved through email, discussion forums and other non-concurrent communications mechanisms, while synchronous collaboration is achieved through real-time chat, teleconferencing (audio / video) and similar technologies.

2. TRADITIONAL CLASSROOM APPROACH

When OMSE was first launched, each course was composed of 3-hour classes one day per week in the evenings over a 10-week period (plus an eleventh "exam week"). Lectures were presented via a data projector; and Internet access made it possible to discuss web content. Course materials were initially distributed in binders and updated as required. Attendance in the classroom was expected and grades were awarded for both class participation and the quality of assignments and take-home exams. Student work was submitted by email for the most part.

Soon, learning materials were made available through a password-controlled web site which significantly enhanced access to learning materials and facilitated updates by the instructor throughout the term. However, access control, updating, version control, and backup were conducted in a fairly ad hoc manner leading to possible inconsistencies and the risk of occasional unavailability of learning materials.

Being working professionals, OMSE students required a workable solution when on travel status, as well as on those occasions when sick or otherwise pulled away from class for personal reasons. We therefore began to record classes onto video tape using a fixed camera and a podium microphone, making the tape recordings available for later viewing. This was a reasonable solution at the time. However, with the fixed camera aimed at the lecture materials projected on the screen, the audio/video production quality was fairly low quality and rather "amateurish" looking. And the logistics of recording, borrowing and returning tapes was an inconvenience for instructors, students, and the Program Office.

3. OUR INITIAL EXPERIENCE WITH ONLINE LEARNING

Open University [1] in the United Kingdom was the first university successfully delivering distance education by means of communications technology. E-learning, also commonly referred to as online learning, emerged out of numerous efforts over the 1970 to 1995 period. Bates [2] comprehensively describes the range of possible technologies for distance education including cost structures, teaching

applications, organizational issues, and course development implications. At about this time the potential of the Internet for e-learning began to become a reality and new online collaboration technologies and pedagogies began to proliferate. The pros and cons of synchronous and asynchronous e-learning are discussed extensively in the literature by authors such as Hrastinski [8].

Learning Management Systems like WebCT [3], Blackboard [4], and eCollege [5] became available in the 1995 to 2000 timeframe. Most of these efforts did not favor synchronous e-learning collaboration using teleconferencing technologies – they were arguably too limited in performance and/or too expensive to deploy and scale for university and college learning applications. Instead, these learning management systems emphasized asynchronous e-learning (a.k.a. Asynchronous Learning Networks (ALN)) that leveraged the web by way of online discussion boards/forums and email. These systems also incorporated repositories for maintaining and dispensing learning materials and related resources to students on demand.

The arguments favoring the ALN have been well articulated by the Sloan Foundation [7] and others. Asynchronous discussions stimulate critical thinking, reflection, and thoughtful (written) expression which enable students to engage more deeply with instructors and other students than in traditional classroom learning. In the case of OMSE, because all of the students are software professionals, asynchronous discussions provide an opportunity for them to learn a lot from each other as well as from the instructor – an additional value-add. OMSE's first foray into online learning considered the following benefits of asynchronous learning:

- Students do not need to come to campus and meet at preset times;
- They can participate online at virtually any time and from any place;
- The flexible learning style allows them to read and think before expressing their thoughts;
- Students reflect more deeply on the many complex issues of the subject matter;
- Students whose primary language is not English can more easily keep pace with the class.

Counter-arguments included:

- Social Aspects such as the potential of increased isolation of students and instructors from each other;
- Barriers to getting acquainted and planning out learning tasks;
- Reduced effectiveness of motivational queues derived from facial expression and body language;
- Issues of academic integrity compromise do to the challenge of authenticating student work (assignments and exams).

Most of these issues have been countered in the literature. For example, Wegerif in [9] espouses the social dimension of asynchronous learning and makes constructive recommendations arguing that appropriate planning and instructor facilitation will build strong communities of learners and effective learning. Building a measure of residency requirement into a program can mitigate the student authentication risk.

Early in the OMSE program we had not anticipated the possibility of putting the entire program online. When the subject came up we wondered whether and how online models would satisfy issues of faculty engagement, belonging, and honesty. With respect to engagement we began to recognize some of the benefits of e-learning, and we began to better appreciate that traditional courses also ran similar risks of student comradeship and academic integrity. Given that our students are working professionals and often work in distributed and ever-changing teams – we argued that such risks were manageable. We concluded that the pros outweighed the cons for our target audience.

In 2000, the OMSE Program Office began to invest in the necessary planning and preparation effort to offer a few courses online. In 2002 and 2003 we converted and offered three of our software engineering courses via the eCollege learning management system thereby increasing learning flexibility and options for students. This initial experience with e-learning set in motion OMSE's gradual evolution of flexible and innovative online and hybrid learning models. Feedback from students and instructors was positive.

However, a commitment to make additional OMSE courses available online was delayed by a few years due to lack of buy-in and lack of sufficient resources. Over time we partially overcome these barriers.

4. CAPTURING AND STREAMING LECTURES AND Q&A SESSIONS

In 2003, OMSE courses began to be offered in PSU's Distance Learning Center (DLC). The primary objective of this transition was to replace video tape recording with web-based streaming.

The distance learning center classrooms contained 3 cameras, push-to-activate microphones, a data projector, a PC, Internet access, a document display device ("Elmo"), and various controls and monitors. Student assistants in a neighboring control room managed the cameras and sound; the instructor controlled the presentation sources; and the students controlled their microphones. This enabled the recording of lectures which where digitized and made available via streaming to students who were unable to attend the face-to-face sessions.

This was a significant improvement over the video tape-recording system for three reasons: it eliminated much of the logistical problem of tapes; the video of the instructor and projected materials were more dynamically captured; and student participation in the classroom was made available via the data stream over the web. However, the quality of lecture slides was rather low - degraded by the digitizing and streaming processes - obliging instructors to use larger fonts than they would have otherwise. And the audio from the student participants was inconsistent because of the need to "push to talk".

Unexpected Benefit: An unintended capability that materialized was that the streamed classes were often, but not always, available 5-10 seconds after being captured. This allowed students who were unable to come to the classroom, but otherwise available, to remotely access and view the a/v streams. Students with other obligations at home, at work, or on travel status could thereby keep pace with the course in close to real time. These streams could also be accessed days later, or whenever students might want to review specific course content.

5. LEARNING MANAGEMENT SYSTEM

In 2004, PSU was routinely supporting WebCT [3], a popular learning management system. This LMS, which was similar to eCollege in several respects, was supported by PSU's IT infrastructure. Between 2004 and 2006, we "resurrected" our initial OMSE online courses by migrating them from eCollege to WebCT. We began to offer these courses online, as well as in the traditional face-to-face learning mode, leveraging WebCT's support for asynchronous discussion forums and assignment submissions.

Guidelines for conducting and pacing online discussions were provided to students and adopted by OMSE instructors. During "discussion weeks" the instructor posts one or two questions. The students are required to first post a well thought-out response to each question by mid-week, and then rebut the primary posts of at least two other students by the end of the week. The instructor comments on every primary post and selected rebuttals first; and then provides a single substantive feedback post to all of the students at or very soon after the close of the week.

WebCT's comprehensive assignment support capabilities were also used for posting, downloading, uploading, tracking, submitting, grading and feedback purposes.

In 2008 WebCT was replaced by Blackboard CE 6 [4]. Conversion of courses from WebCT to Blackboard was not painless. Conversion was successfully achieved by way of the devoted efforts of the instructors. In 2010-11, PSU replaced Blackboard with Desire2Learn [6] and a similar conversion effort was required. In summary, online OMSE courses progressively exploited the following learning management capabilities:

1. Access to course resources including syllabus, lecture slides, articles, and relevant web links;

2. Weekly lesson plans pointing to assigned resources, activities, discussions and assignments;
3. Discussion forums supporting asynchronous collaboration among students and instructors;
4. Weekly assignments, take home exams and quizzes with submission guidance;
5. Grading ("Gradebook") support allowing instructors to record, track, and process grades;
6. "My Grades" – students are able to view their grades from week-to-week;
7. Announcements – instructor messages to all students that pop up when they log on;
8. Email – students are able to capture all student and instructor emails related to the course;
9. Student Lists – enrollment status;
10. Student Usage Tracking – # of accesses to assignments, discussions and other elements.

6. PRE-RECORDED AND NARRATED MINI-LECTURES

When OMSE was resurrected in the 2004-2006 timeframe, we acknowledged the above-mentioned criticisms of asynchronous learning, in particular, that students may tend to feel isolated from the instructor and each other. We attempted to partially overcome this problem by posting:

a) Pre-recorded mini-lectures or
b) Narrated PowerPoint mini-lecture presentations.

Mini-lectures are meant to highlight the most important points, pace the course, and tell the students where to look for the details (e.g. in textbook, articles, etc.). The strategy we adopted depended on the following assumptions:

- That weekly full-lecture (3-hour) presentation slides are available
- That the instructor is very familiar with these slides, and
- That the instructor has already created 20-30 minute mini-lectures (10-15 slides each) from them.

Our experience is that:

i) Pre-recorded mini-lectures require the instructor to invest an estimated 1.5 to 2 hours per mini-lecture in the studio presenting the mini-lecture slides in front of a camera.
i) Narrated PowerPoint mini-lectures require about the same investment in the instructor's time but do not require the use of a studio and camera-person.
ii) Narrated lectures are easier to update. One can add, delete and re-narrate individual PowerPoint slides without re-recording the entire mini-lecture. This is impractical with studio-recorded mini-lectures, or at least considerably more challenging and costly;
iii) Narrated mini-lectures can become rather large and impractical for students to download. One must keep the narration crisp; avoid silences and pauses; and use appropriate recording settings to keep volume of digitized narration from becoming too large.
iv) Studio-recorded mini-lectures can ramble on more in an attempt to squeeze in more content, however, the idea is to be brief and to the point.

We have had mixed feedback about the benefits of such mini-lectures. Additional study is required in this aspect of our approach. We continue to be hopeful that this type of learning element will be useful if augmented by other elements that help engage the students and the instructor.

7. LEARNING SYSTEMS SUPPORT F2F

Prior to the availability of WebCT and Blackboard, OMSE courses maintained course content on a password-controlled web site. Instructors were responsible for maintaining / updating these materials. With the introduction of commercial learning management systems it became possible to maintain learning materials in a structured course container which provided a common organization with stronger, more flexible, and centrally administered access controls, maintenance, and a help desk.

Although WebCT and Blackboard were somewhat “clunky” in comparison to commercial tools, they are relatively intuitive and easy to use. This was definitely a significant step in the right direction. This greatly enhanced instructor productivity with respect to syllabus, lecture and assignment preparation; grading, class coordination, and student usage tracking; and the setup and engagement in online discussions. Progressively, our instructors learned how to exploit these useful functions and features.

I observed that it would be a relatively small step for an LMS-enabled traditional face-to-face course to add a few online discussion forum assignments and thereby partially resemble an online course. The distinction between traditional and online learning modes was becoming increasingly blurry.

8. OUR INITIAL EXPERIMENT WITH HYBRID LEARNING

In 2007 we began to consider the possibility of blending face-to-face and e-learning delivery modes. Although certain of our students preferred face-to-face sessions over online courses, we reasoned that their busy schedules would make online collaboration an attractive alternative when students are unable to come to campus. And we knew from experience that many students are sometimes unable to attend for various logistical and personal reasons. So we asked ourselves, why not let the students mix and match elements of face-to-face and online delivery modes? Wouldn’t this potentially maximize the distinct benefits and provide more flexibility for our students?

We began by offering our professional communications course to a number of students who participated strictly in online mode - never coming to class. They received delayed streams of the face-to-face sessions attended by the other students in the class and participated in the same asynchronous discussion forums as the face-to-face students. Although not present in the classroom, they were an integral part of the collaborative learning process, receiving feedback and grades just like everyone else. This initial offering really tested what could be effectively learned through the combination of asynchronous e-learning and streaming media.

This first hybrid delivery-mode course opened our eyes to the possibilities and benefits of offering combined hybrid face-to-face and e-learning courses. We soon realized that software professionals are already accustomed to working with each other online. For them, sharing information via email, instant messaging, VOIP (e.g. Skype), bug-trackers, version control systems, document sharing tools, and collaborative web pages (wikis) is routine. The jump to online e-learning is a small hop for them and quite natural. But most professionals also want to get some of the learning from the “horse’s mouth” -- an instructor they can interact with in the classroom.

Our observations validated our hybrid vision for learning, that is, one where we simultaneously deliver learning face-to-face and also collaborate over the web. We reasoned that this would achieve superior results by combining the two approaches (Milhauser and Toth [10]). Such a flexible approach to learning certainly provides distinct advantages for the student - they can decide whether they would rather take the course face-to-face, or online, or in some combination. For students unable to come to campus, the online delivery mode remains a viable option.

9. UPGRADING STREAMING QUALITY

In 2009, a product called “Echo 360” was introduced to upgrade the quality of delivered streams. This tool captures and mixes the audio-video feed from the remotely controlled cameras in the classroom with the data presentation, document camera, and Internet browsing sources. The technology broadcasts this instructional material to learners, just-in-time or delayed, through a highly flexible and media-rich user-controlled interface. These streams are made available through links automatically embedded into the learning management system. The quality of the presented course materials and a/v streams was significantly improved by this transition to Echo 360.

10. THE INTRODUCTION OF SYNCHRONOUS COLLABORATION

Bates [2] and other authors have long espoused the use of teleconferencing tools for synchronous online learning. Such tools had the potential of increasing social, interpersonal, motivational, and organizational aspects of the online learning experience. And such synchronous capability complements the proven benefits of asynchronous engagement and learning. Chou [11] illustrates the benefits of synchronous learning integrated with traditional ALN systems and makes useful recommendations that leverage "student-moderated" synchronous sessions. As already mentioned, the high costs of these tools put them out of reach for universities and colleges in the early years of e-learning.

However, great advances have been made more recently. Internet bandwidth has increased in leaps and bounds at very affordable levels for both universities and students. And a plethora of comprehensive webinar and teleconferencing tools have become available. Of particular interest is the paper by Latchman et. al. [12] that presents a comprehensive framework for such synchronous mechanisms that have been apparently borrowed and adapted by many of these products.

In the fall of 2009, PSU acquired licenses for a high quality synchronous collaboration tool called "Elluminate" [13] which supports both webinars and teleconferences. Elluminate's long list of functions and features include session scheduling and notices, audio conferencing, shared visual space, text messaging, webcam support, and session recording. We found the tool to be acceptably reliable and scalable.

During 2009-2010, we began to phase in this tool to support synchronous team meetings for our practicum course, several synchronous Q & A sessions for our software estimating course, and a couple of webinars.

11. THE HYBRID LEARNING PROCESS

Under our hybrid learning approach, attendance at face-to-face sessions is optional. Some students will attend most of these F2F sessions; some will never attend (online-only); and still others will choose to attend occasionally.

All students make use of a common pool of learning resources (textbook(s), articles, papers, hyperlinks to electronic documents, etc.). These resources are made available through our learning management system (LMS). Through a weekly lesson plan posted in the LMS, learning is synchronized for both face-to-face and online learners. Student activities are guided through a checklist that may include mini-lectures highlighting key points, posted full-up lecture materials, textbook and other assigned readings, and posted take-home assignments. Discussion forums are extensively used to asynchronously engage all students and instructors in thoughtful exchanges of ideas and exploration of assigned problems.

The cadence of student learning through assignments, discussions, and project work is synchronized across the face-to-face and online dimensions of a course. One approach we have used is to hold an initial discussion in the classroom (which is streamed), summarize the results, and then post them online to seed a more in-depth discussion during the week. For example, students in the face-to-face class on a Tuesday night might conduct a discussion of the benefits and challenges of software quality and code reviews; and then explore this issue further in the online learning space with an asynchronous discussion thread created and facilitated by the instructor. Conversely, an asynchronous online discussion might be held first, with a summary discussion in the classroom.

Regardless of the exact process or timing, the instructor must ensure that students attending classroom sessions, and those participating online, are active participants of the larger integrated discussion.

12. LEVELING THE PLAYING FIELD

To achieve assessment equity across a hybrid model, grading of assignments, discussions, participation, and project work need to be balanced and fair - whether the student is face-to-face, online, or both. Because learning is accomplished differently, achieving such balance requires the careful attention of the instructor.

In the classroom setting, students traditionally get "participation points" for showing up in a classroom and the instructor tries to get a measure of the quality and degree of participation of all students in the class. The more extraverted students stand out and often make substantial contributions to the class. But some student contributions are more “noise” than substance and in some cases may actually be disruptive to the class. Meanwhile, introverted students or those less proficiency in English will be less visible or not visible at all. It is clearly not practical for the instructor to count the number of times a given student speaks out, or measure the quality of each contribution fragment of the student. Because of these limitations, most instructors tend to allocate a small proportion of the total grade for in-class participation and use a rather subjective and low granularity assessment scale (e.g. poor, fair, good, excellent) of student participation.

In an online class, meanwhile, introverted students can shine. Those with trouble in English can take extra time to express themselves, and the more extraverted students, especially the “noisy” ones, are obliged to express themselves more clearly and thoughtfully than they would in a face-to-face setting. The main advantage of online versus face-to-face discussion is that the instructor can actually assess the participation level first, and then examine the true quality of written student postings. The advantage is that the quality of online participation can be assessed with more assurance and evidence than the quality of participation in classroom sessions. This allows a larger proportion of grades to be allocated for online participation.

A critical factor to achieve balance is to ensure that both face-to-face and online students participate at the same level in online discussions. It is important to distinguish between the participation level in online discussions and the quality of these discussions. In-class participation grades should be eliminated altogether when conducting such hybrid delivery-mode courses.

13. TEAM PROJECTS

Whether accomplished through face-to-face or e-learning mechanisms, team projects are always a challenge. Projects run in a classroom setting generally have the students meeting in classroom time slots, sometimes with the instructor present judging team participation. Of course, team presentations are a key component of assessment. Elluminate has proven to be very effective for supporting synchronous online sessions among students working in teams, including project team presentations [14].

Assessing team participation among members of a distributed project team is a bigger challenge. Often teams will work on their own using their collaborative tools of choice. Some may use simple email with scheduled status updates provided to the instructor. Others will use free tools like Google Docs and Yahoo Groups. At PSU we have had good success with both PSU Online (WebCT and Blackboard) and Wiki collaboration tools. They each have their advantages - but also their weaknesses. The instructor becomes a member of every team to provide oversight and guidance, and can monitor the quality of team collaboration. Online tools thereby facilitate project team assessment.

14. LEARNING ASSESSMENT AND CONSISTENCY

At this stage I wondered how we could level the playing field between face-to-face and online (only) learners - those who did not attend many if any of the face-to-face sessions. We found no easy answers. One level-setting practice that I have used successfully is to create assignments that require students to summarize their learning (whether online or face-to-face) at key points in the class. Another is to conduct peer assessments where all team members submit the evaluations of how well each of their teammates

contributed toward their shared goals. Again, the same assessment method is used whether the class is face-to-face or online.

A key to success is to ensure that instructors are comfortable with the level of engagement, whether the student is online or face-to-face, and that expectations of students are spelled out clearly and reinforced repeatedly throughout the course.

15. ENGAGEMENT AMONG FACULTY AND LEARNERS

In addition to learning extracted from course content and resources, students learn from the instructor as well as from each other. This is particularly true of courses offered to experienced practitioners such as those in the OMSE program. An important goal of ours has been to engage students and instructors via our blended (hybrid) delivery-mode and e-learning formats. The following scenarios represent approaches we have used to engage face-to-face and online learners with the instructor and each other:

a) Studio-recorded lectures or lectures narrated by the instructor in advance. They are accessed through the learning managing system by downloading them or by streaming.
b) Questions posted by the instructor to stimulate online discussions related to the week's subject matter conducted asynchronously within a defined time interval.
c) Weekly Q & A sessions facilitated by the instructor and streamed to the online students. The instructor summarizes the results and challenges the class with related questions to be discussed online asynchronously.
d) Synchronous sessions run in lecture style (e.g. like webinars) or interactive Q&A sessions. By using tools like Elluminate, the need for face-to-face sessions can be significantly reduced or even eliminated.

16. FISCAL AND PHYSICAL IMPLICATIONS

I began to consider the cost and space implications of hybrid course offerings. By shifting more of our learning to asynchronous and synchronous e-learning collaboration mechanisms, there is a strong argument for reducing the number of face-to-face hours required to satisfy learning needs and student preferences.

Being working professionals, OMSE students are increasingly telecommuting and working in distributed teams – often dispersed regionally and even globally. They are using asynchronous and synchronous processes. For example, software practitioners already make extensive use of email, version control systems, problem tracking and resolutions systems, shared knowledge databases, and team blogs – all of which are asynchronous. And many are also beginning to leverage teleconferencing and webinar tools which are synchronous. The physical space they need to work synchronously with each other is rapidly shrinking.

Similarly, hybrid learning can reduce demand on classroom space by increasing the learning elements that are conducted online – both synchronously and asynchronously – thereby decreasing the amount of face time required. Our initial guess is that face-to-face time can be reduced 50% without impinging on the needs of those students expecting and/or demanding face-to-face learning. Delayed streaming and studio and narrated mini-lectures may be sufficient to satisfy students who are either more interested in online learning, or students who cannot attend. Recorded synchronous Q & A sessions and webinar-styled mini-lectures may also be very acceptable to a significant number of students. And of course, streams and recorded synchronous sessions will prove to be adequate for all students to cover absences from the classroom.

17. SUMMARY

Students who have taken hybrid courses have reported that because they can schedule their class time to dove-tail with their other day-to-day obligations, they are able to spend more time participating in online discussions and individual research – both of which increase the depth of learning. And it is common to receive reports of deeper engagement with classmates, instructors, and subject matter through online collaboration. At the same time students can elect to attend only targeted face-to-face classes to engage more closely with specific areas of interest and need. And the availability of streamed and recorded synchronous sessions will satisfy the social and motivational needs of many students.

From an institutional budget perspective, hybrid courses reduce the required classroom time thereby liberating the use of the space to other classes. This increases overall classroom space utilization and decreases operational costs for the university.

The net result is better total learning outcomes, improved retention, and flexible learning models that better integrate learning with work and home life – this is exactly what working professionals need and want. And better student retention is a critical success factor for the university as well as our program.

REFERENCES

[1] Open University, http://www.open.ac.uk/

[2] Bates, A.W., "Technology Open Learning and Distance Education", Routledge Studies in Distance Education, 1995

[3] WebCT, http://en.wikipedia.org/wiki/WebCT, a web-based learning management system originally developed at UBC and later acquired by Blackboard

[4] Blackboard Learning System: http://www.blackboard.com/, a web-based learning management system owned by Blackboard Inc.

[5] eCollege, http://en.wikipedia.org/wiki/ECollege, an on demand hosted learning management system now owned by Pearson PLC

[6] Desire2Learn, http://www.desire2learn.com/, a hosted learning management system

[7] The Sloan-C International Conference on Asynchronous Learning (http://www.aln.ucf.edu/)

[8] Hrastinski, Stefan, Asynchronous and Synchronous E-Learning, Edcucause Quarterly, No. 4, 2008.

[9] Wegerif, Rupert, The Social Dimension of Synchronous Learning, Journal of Asynchronous Learning Networks (JALN) Volume 2, Issue 1, March 1998

[10] Milhauser, K, Kal Toth, Better Mileage with Hybrid Learning, Software Assoc. of Oregon, June 2007

[11] Chou, C. Candace, A Comparative Content Analysis of Student Interaction in Synchronous and Asynchronous Learning Networks, Proceedings of the 35th Hawaii International Conference on System Sciences, 2002

[12] Latchman, H.A. Ch Salzmann, D. Gillet and H. Bouzekri, Information Technology Enhanced Learning in Distance and Conventional Education, IEEE Transactions on Education, November 1999

[13] Elluminate, http://www/elluminate.com

[14] Toth, Kal and Raleigh Ledet, Lessons Learned About Software Team Collaboration, Pacific Northwest Software Quality Conference (PNSQC), Oct. 2010.

[15] Toth, Kalman C., "Software Engineering Online and Hybrid Learning Models at PSU", International Conference on Computers and their Applications, New Orleans, March 2011

[16] Toth, Kalman C., "An Organizational Approach for Sustaining E-Learning in a Large Urban University", Future of Education Conference, Florence, Italy, June 16-17, 2011

Hard Lessons About Soft Skills -- Understanding the Psyche of the Software Tester

Marlena Compton
mcompton@mozilla.com

Gordon Shippey, MA, LAPC
gshippey@gmail.com

Abstract

Testers are often characterized as the conscience of a project. This session will dive deeply and uncomfortably into the psyche of software testing and those of us who claim it as our profession. Attitudes that are deeply embedded in our testing culture and their effectiveness will be examined:

1. Our bug finding mandate is a wrecking ball, which we aim and swing at will
2. We haven't done our jobs until the developers are crying and the ship date has been extended by months
3. Developers live to write bad code and should be eyed cautiously at all times
4. Crunch time on a project is our signal to be as harsh and unforgiving as possible to everyone else on the software team.

The positive role that emotions can play in testing is also discussed. Crucial conversations are also introduced to help testers understand how to argue their point without bullying. This presentation will show why testers must have and act with conscience and have an emotional fluency in order to test software effectively. We will show that despite being erroneously called soft skills, the effective care and handling of emotions is the hardest skill in testing.

Biographies

Marlena Compton has been testing software since 2007 and has an innate talent for making developers feel angry, inadequate and ashamed. After noticing that happy developers make better software, she embarked on a journey to learn how to be an effective tester and help the developers feel supported, at the same time. This led her to research psychology and communication skills as they apply to testing with friend and licensed counselor Gordon Shippey.

Gordon Shippey has completed degrees at Emory University, the Georgia Institute of Technology, and Argosy University. A Licensed Associate Professional Counselor in the state of Georgia, Gordon provides psychotherapy to individuals, couples, and families. Gordon is particularly interested in work/life balance and organizational psychology.

1. Introduction

One of the main ideas in the Agile Manifesto is that individuals and interactions are valued over processes and tools. This has brought a new dimension to software development especially now that ideas from agile software development have become mainstream. People are encouraged to talk and work more closely with each other. Whereas before, everyone had their own area of working with the software, people on software teams engaging in agile software development have to figure out a way to work closely with each other. Office plans suggested for agile teams include the removal of cubicle walls which means that if people are "faking it 'til they make it" in trying to get along with each other since there is no place to hide.

For testers, it means that if a tester uses software and hates it, developers can now literally hear the screams. If a developer talks about putting off bug fixes in a standup, and it makes a tester cringe, everyone on the team sees it. Such knee-jerk, emotional reactions are important in testing. It is important for a tester to develop a sense of how an application will fail. Unfortunately, the reactions spurred by tester instincts can be also be damaging for a team's morale.

While it is extremely easy to make these mistakes and perpetuate bad attitudes (the authors have first-hand experience), the skills needed to avoid them are not easy to come by naturally or even easy to learn. Examining this failure in tester communication and attitude is difficult and uncomfortable, but necessary. Some of these attitudes are ugly and therefore may be denied or resisted, but ignoring them allows them to linger on. The authors hope that testers will see this paper as an opportunity for self-reflection and growth instead of a simplistic attempt at painting testers as villains.

There are few places in the testing literature to turn for answers if a tester is having problems communicating with developers on her team or if she is feeling ostracized because of communication mistakes she has made. On the contrary, the literature on skills required for software testing is well established. Much has been written about techniques that can be used to analyze software. Testers have any number of resources that they can choose from, such as *How to Break Software* or *A Practitioner's Guide to Software Test Case Design* if they wish to improve their test case design or to learn how to break software in any number of ways. We are, however, missing out on a foundational skill in software testing. Testers must learn how to find the balance on their team between providing criticism of the software but remaining constructive in the process.

This paper will borrow from recent writing on workplace interaction and place it within the context of software and software testing. Books such as Dr. Robert Sutton's *The No-Asshole Rule: Building a Civilized Workplace and Surviving One that Isn't*, and *Crucial Conversations: Tools for Talking When the Stakes are High*, by Kerry Patterson et al. is used to describe the social and emotional skills necessary for software testers to do their work while remaining constructive members of the software team.

We will begin in Section 2 by discussing why a work environment that is psychologically safe results in better software. With the need for a safe workplace established, we will move into describing negative behavior and what it looks like when placed in the context of software testing in Section 3. This behavior forms the basis of what is described in Section 4 as "The Tester's Paradox." "The Tester's Paradox" explains how testers are set up to fail at communicating effectively on any software team. With the need for better communication skills from testers established, the next sections describe in a practical manner, fundamental skills testers can use to improve their communication foundation. Crucial conversations are

introduced in Section 5 as they apply to software testing. The physical barrier to effective conversations is examined in Section 6. This is followed in Section 7 by methods for creating physical and conversational space in software testing. Section 8 links the concept of mindfulness with software testing as a method for working more effectively with our emotions when we are testing. Finally, Section 9 explains how we can reach out when things go wrong.

2. Why psychological safety means better software

Building software requires creative thinking even if the budget is tight and the software must be turned around quickly. In their book, *Artful Making*, Rob Austin and Lee Devin write about how the creative work in software takes place on an edge. It requires a team to stretch itself uncomfortably and should allow for frequent failure. In such an atmosphere, having a high overhead for failure will block creativity necessary for making software. Working on the edge cannot include fear or the quantification of work, but must include practice and self-knowledge.

It is important for the members of a software team to be able to argue constructively in order to work out complex problems creatively. Unfortunately, it is easy for arguments over software to descend into personal insults and ill will. This damages a project's safe atmosphere and means that team members will be less likely to discuss their ideas.

People are damaged in unsafe workplaces. Sutton describes this damage in *The No Asshole Rule*. The immediate effect on a person can include heightened anxiety, heightened depression and trouble concentrating. Negative interactions are felt five times more strongly than positive interactions. They affect not only the target of the insult, but also have a ripple effect on the rest of the team. Teams experiencing high levels of negative behavior become fear-driven, with everyone looking over their shoulder.

A good indicator of safety is what happens when people make mistakes. In his book, *Good Boss, Bad Boss,* Sutton suggests that failure can be handled in any of three ways. If a tester keeps a wall of shame on their cubicle wall showing bugs they found particularly offensive and the name of the developer responsible, the tester is engaging in a common and very destructive reaction to failure described by Sutton as “the Silicon Valley standard.” The tester is simultaneously remembering, blaming and shaming developers for bugs assigned to them. Another less destructive but undesirable method of handling failure is forgiving and forgetting. This is where the line is drawn between being effective as testers and being “the push over” on a team. It might make the developers feel better momentarily if a tester allows them to ignore bugs that are minor. This can especially be the case if a deadline is close and the testing schedule is extremely tight. It is essential to remember that a tester’s job is to report when things go wrong.

Rather than being overly harsh or too forgiving, a tester can, as part of their team, practice what Sutton calls, “forgive and remember.” If it appears that a developer has not written code that matches a specification or user story, it is worth asking the question of whether the team communicated that user story effectively. Since the team acknowledges the mistake, the team can now learn from the problem. No fingers are pointed. This can be a lesson for developers as well. If a tester has failed to find major problems in a release, the tester’s team should acknowledge the failure and do what they can, together,

to understand what went wrong. This turns the failure into a learning opportunity for everyone rather than isolating the blame on one person.

One cost of a work environment based on fear is the increased time spent on finger pointing instead of fixing problems or even reporting them. In a study conducted at Harvard Medical School, nurses working in units where they felt psychologically safe reported much larger numbers of errors. In these units, mistakes were seen as natural and reporting them as just part of a normal day's work. In units where nurses did not feel safe, the number of errors reported was much lower because the nurses were afraid of the repercussions that would result, not because they were less error prone.

One of the authors of this paper has seen the effect of a fear driven environment when she worked with developers who had bonus pay docked every time a bug was found in their code. This led to discussions in which developers would “bargain” with testers to try and do anything they could to get out of having a bug filed against their code. This was in stark contrast to the atmosphere at a different employer where developers would seek out the tester and tell her their suspicions about how their code might break. This opened the door to constructive dialog between the developers and the tester.

3. Negative behavior in the context of testing

Before we delve into the ways testers can strengthen their communication skills, we will examine negative behavior and what it looks like in the context of testing. This is not intended as an insult to testers but is merely intended to give testers an opportunity to reflect on how they may be sabotaging their own best efforts to contribute to discussion on their software team.

Hurtful or negative behavior is familiar to everyone. No one escapes life without experiencing insults, bullies and toxic relationships. Sutton examines this behavior as it applies to the workplace in *The No Asshole Rule*. He points out that the damage suffered is not large and dramatic but rather through an accumulation of many small, demeaning acts. His list of negative behaviors include:

- Personal insults
- Threats and intimidation
- Sarcastic jokes and teasing as an insult delivery system
- Overly dramatic emails containing harsh language such as words written in capital letters
- Public shaming
- Interruptions
- Nasty looks

Bugs are easily used as insult delivery systems or for delivering our own personal judgments about functionality. Testers should be sensitive to how they write bug reports since they can last for a long time and are often read in many different contexts.

If testers feel they are not being listened to, they may result to public shaming such as pointing fingers in a standup. Threats delivered in emails are an unfortunately easy way to amplify a message we may feel is not being heard. One of the authors is particularly guilty of using nasty looks to let developers know when she was displeased by a feature or decision.

Finding most of this behavior in almost any software team is easy, and it is not just testers who participate. Indeed, the culture of software has thrived on sarcasm and jokes based on insults such as the humor found in popular web comic, *The Oatmeal*. Testers are no exception. In fact, tester hazing of developers is a typical source of “tester humor.” It can be difficult, however, to know when the humor ends for someone, and the danger to psychological safety begins.

4. The Tester’s Paradox

In *The No Asshole Rule*, Sutton points out that anyone can be a jerk given the right working conditions. To place negative tester behavior in context, it is necessary to consider working conditions for the average tester. If the tester is on a team using a waterfall approach or a team lacking the benefit of continuous builds the tester may spend significant amounts of time being blocked by problems further upstream in the software process over which they have no control. When the work finally reaches the tester, it is often already late and over budget. This means the tester will be critiquing software from a team whose nerves are already frayed.

If a tester is working on an agile team with continuous builds, they will have more control and autonomy over when they can do their testing, however, they may not be sensitive to the fact that the developers are working on their creative edge. This causes further sensitivity on a team.

While some testers are fortunate enough to work on a team with a low developer to tester ratio, many testers work on much smaller teams or even as the solo tester. When a tester is working alone or lacks enough support, it can be challenging to be heard. In environments where developers are encouraged to be "cowboys" it is even more difficult for testers to have a balanced relationship with the rest of the team. This can lead to the tester feeling ostracized and powerless. Even when testers work on larger teams, they may live in a silo with very little opportunity for communication with their developer counterparts. Testers can be separated from the development team by being in a different building, a different location or a different time zone. In this case, the tester lacks a connection to the rest of their team making it a struggle to feel that they have any voice at all.

Aside from looking at the environment in which testers work, it is worth considering what testers do. Testers must critique an application. This includes asking probing questions, finding ways to break an application and reporting bugs about the breakage. In the case of user interface testing, testers must report if something about the user interface makes it awkward to use. The basis of all of these activities is finding information about how an application is broken, not working as intended or confusing to a user. All of this information is negative.

The reality of testing is that the task of producing negative information about an application is given to the person on a team who may feel ostracized, powerless and alone. This creates an unfortunate situation. Testers are placed in the most emotionally precarious position of anyone on a software team and blamed if they don't handle the situation correctly every time. This is the tester’s paradox. Just as each and

every negative interaction packs five times the punch of a positive one, a study has shown that it takes approximately 5 interactions to get over one bad interaction. 100% perfection is not a fair request to make of any human being, including software testers.

5. Crucial Conversations in Testing

In, *Crucial Conversations*, *Tools for Talking when the Stakes are High*, Patterson defines a crucial conversation as, “a discussion between two or more people where 1) the stakes are high 2) opinions vary and 3) emotions run strong. Software testers face these types of conversations every day in a myriad of different situations:

- Discussing whether or not a discovery is a feature working as designed or a bug
- Participating in a discussion about which bugs should be fixed before a release
- Asking whether or not it is ok to ship a release
- Discussing why a bug is reproducible in one environment and not another
- Discussing the flexibility of release dates
- Asking a developer if a feature is ready, especially if that feature is late

Crucial conversations can happen at any time, and in the compressed, intense nature of agile iterations, they happen frequently and are most often spontaneous. Opportunities for crucial conversations in an agile project include post-standup conversations, retrospectives, and any daily interaction. Agile office spaces are designed to encourage team interaction and discussion. Sitting at a desk on an agile team frequently means that there are no walls and the rate of discussion is much higher. The purpose of this is to create and maintain a pool of shared meaning on the team. While this is a great way for testers to be more integrated with their team, it also creates a challenge for testers, along with everyone else on the team, to have a net positive effect in the many conversations they will have.

The beginning of any crucial conversation is, what Patterson calls, “starting with heart.” When approaching any conversation, it is necessary to know what you want from it. This can be as simple as wanting to know if we have gotten the correct meaning from a story or as complicated as having to answer the question, “is this software ready to ship.” Starting with heart is important because no matter where the conversation goes or how messy it gets, it helps us to retain focus on what we wanted in the first place. If we don’t know what we want from a conversation, a simple, “let me get back to you on that,” should be acceptable. If it is not, someone in the conversation will begin with an unfair advantage and creative safety will likely be compromised.

There are a few questions to ask before engaging in a crucial conversation:

1. What do you really want for you?
2. What do you really want for the team?
3. What do you really want for the software?

Once you’ve answered these questions about what you want, it is also worth asking yourself how you would behave if you really wanted these results. This will help you to focus on your goal and get your brain ready for the conversation. It is worth noting that if the only good reason you can come up with for having or staying in conversation is to one-up someone else or to “win,” it is time to reassess your true goal.
Sometimes crucial conversations take the form of arguing. When a developer insists that software “works in her environment” or a tester is faced with answering the question, “is this software ready to ship.”
In his book, *Good Boss, Bad Boss*, Sutton lists a few guidelines for having a constructive argument:

- “Don’t begin the fight until everyone understands the challenge or problem at hand.” This is in the same vein as starting with heart. In software, it is particularly important to know that everyone is arguing about the same problem. Even if the discussion is about “feature x”, the argument might be going in the wrong direction because each person is arguing about a different nuance of “feature x.”
- “Don’t argue while generating ideas or solutions - make it safe for people to suggest crazy or controversial ideas.” For testers, this means holding back from torpedoing an idea too early or in the wrong forum. Common forums for brainstorming include developer “spikes” or having a “lab week” where everyone works on something new and experimental.
- “Once the argument is resolved, make sure that the conflict and criticism ceases. It is time to develop and implement the agreed upon ideas. If compromise is reached on bugs that will be fixed for a release, it is not helpful to continue complaining about the ones that have to wait until a bug fix release.
- “After the fight is over, do some backstage work. Soothe those who feel personally attacked and whose ideas were shot down.” it is important to know when to reach out to a developer or product manager who may feel beaten by the QA stick. Reaching out is addressed in it is own section of this paper.
- “If people turn nasty, take a time-out and ask them to turn off the venom. Pay special attention to comedians who deliver devastating insults via jokes and teasing.” Although this guideline is written for a manager, it is important, for us, as testers, to know when our stress level and a team’s stress level has hit it is limit. Cultivating this awareness and using different approaches to create space is addressed in the next section.

6. The Physical Challenge of Crucial Conversations

There are good reasons why a crucial conversation is challenging. The primary reason is biology. The human brain was not designed for having successful conversations in stressful situations. In any stressful situation, the brain will send us signals to do one of three things: fight, flight or freeze. The moment we perceive that a conversation has turned crucial, the flow of blood to our brain is decreased while the flow of blood to our arms and legs is increased. Instead of engaging in reasoning, the brain physically triggers us to try and “win” the conversation or to end it.

Aside from biology’s attempts to sabotage a high stakes conversation, our own skills can also lead us astray. Negotiating stressful conversations is not taught in most academic programs. Most people learn conversational skills from their parents who may or may not have had a great skill set. Because these skills are not intuitive, a lack of training can render a person defenseless.

Despite human biology and despite a lack of training, good intentions help set the foundation for a good discussion. Bad intentions set a course of disaster for a conversation. If the conversation is high stakes, the effects can be devastating. We must always be aware of the reasons for having the conversation as well as the other party’s reasons for having the conversation.

7. Creating Space in Crucial Conversations

Part of knowing how to successfully navigate crucial conversations involves developing an awareness of when we need some space in a conversation. Physical signals come from our body responding to stress. Each person’s body responds to stress differently. For some people, they will feel stress in their

shoulders while others will feel it in their chest. Cultivating this self-awareness means that when we are faced with a stressful decision we will be able to recognize that and give ourselves the space we need to make the decision.

A conversational sign that it is time to make some space is what Patterson describes as the “the sucker’s choice.” A tester who finds something that will negatively impact customers and but is told that it is a feature or “working as designed” faces the sucker’s choice. A classic sign of the sucker’s choice is being presented with two choices that are equally bad as the only two options. On the one hand, if the tester agrees, then software will be released that might have a negative impact on the customer. On the other hand, it is implied that the tester is being a nuisance to the team by suggesting that there is a problem at all. “Those offering up a Sucker’s choice,” writes Patterson, “either don’t think of a third (and healthy) option - in which case it is an honest but tragic mistake - or setup the false dichotomy as a way of justifying their unattractive actions.”

Patterson offers a strategy for setting up new choices in these situations.

- “First, clarify what you really want.” In the case of our example, we don’t want to release software that will negatively impact the customer.
- “Second, clarify what you really don’t want.” We don’t want to waste anyone’s time by fixing something that won’t matter to the customer.
- “Third, present your brain with a more complex problem...combine the two into an and question.” In our case, we might say, “I don’t want to release software that will negatively impact the customer and really don’t want to waste anyone’s time with a fix that won’t matter to our customer.

In our example, the conversation could be shifted finding out how important the potential problem will be to customers.

8. Mindfulness in Testing

Mindfulness is a concept found in many of the world’s religions and has made its way into modern psychology through mindfulness based stress reduction. According to the Bob Stahl in *A Mindfulness Based Stress Reduction Workbook*, It is a practice of, “being fully aware in the present moment.” This involves developing our ability to acknowledge our emotions and suspend judgment in what we observe. Mindfulness training involves learning how to acknowledge our emotions in much the same way we acknowledge what we see and hear without allowing them to completely take over our actions.

As an example, a tester finds that a web page he is testing does not allow navigation with the tab key. “I should be able to hit tab,” he thinks. Since the tester is practicing mindfulness, he acknowledges that he is frustrated and that this frustration is his personal judgment. If he is doing exploratory testing, he begins thinking more objectively about the frustration and what it was about the application that caused him to feel that way.

Anytime we preface a statement with “should,” or “I feel that” we are about to express a judgment. These have their place in testing, but it is important to understand the difference between making judgments and observations. Having an awareness of the emotions leading us to making judgments can also give us clues about our own inner rules. Knowing about our inner rules, especially for testing, allows us to break out of them.

While it is important to be able to observe our emotions when we’re using an application, it is also important to have some outside perspective on our emotions and to be able to balance them with the goals and priorities of a software team. Learning about mindfulness based stress reduction is one way to learn how to acknowledge our emotions without letting them get the better of us in our testing.

9. Recover and Repair

It is inevitable that as testers we will make mistakes and steps on toes in the course of our work. We are, after all, human. Mistakes happen. Sometimes we are overly harsh when we critique work that took someone a long time or we judge a feature to be lacking before the developer has had a chance to finish it. It is important to know how to reach out and work on repairing the damage with what relationship expert John Gottman describes as, “repair attempts.”

Sutton outlines steps for making an effective apology when it is needed.

1. Take the blame fully. In the case of judging to harshly, we might say, “I’m sorry that I was so harsh in my critique of your feature.”
2. Take control over what you can. In the case of our example:
3. Explain what you’ve learned
4. Communicate what you will do differently: “I will be more mindful of my critique in the future.”

Testers are paid to find flaws and in doing so, we will inevitably “break some hearts.” It is important however, to keep in mind that bad feelings can linger beyond the current release. A study pointed to in the “No Asshole Rule” claims that one negative interaction equals 5 positive ones. It is, therefore, worth putting forth the effort to find the 5 good interactions to offset the times when a negative one happens. This corroborates what Gottman says about the effectiveness of repair attempts.

The effectiveness of repair attempts, says Gottman, has less to do with the format of the repair attempt and more to do with, “how much emotional money in the bank you [sic] have with that person.” It has, indeed, been our experience that positive reinforcement can go a long way towards a better relationship within a team. If a developer produces a feature that has problems, but is, nonetheless, an exciting feature, it is worth telling the developer that despite the problems you expect them to fix, we are excited about the feature.

10. Conclusion

The consequence for software testers of bringing agile into the mainstream is that we can no longer afford to ostracize ourselves on a software team and expect to succeed at testing. Bad tester behavior can shake a software team’s confidence and reinforce the “us vs. them” mentality between testers and developers. This rift will, eventually fracture a software team.

We have shown why it is important for software testers to contribute to a psychologically safe workplace environment and which types of tester behavior are counter-productive to this effort. In exploring the “tester’s paradox” we have shown the challenge faced by testers in overcoming negative tester behavior patterns. To help testers re-frame their interactions on a team in a more positive way, we’ve applied the crucial conversations mindset and what can go wrong with them. We’ve offered guidelines for arguing constructively and introduced mindfulness as a way to clarify our emotions and judgments in testing. For those inevitable times when things might go wrong we’ve suggested ways to recover and repair relationships. It is the sincere hope of the authors that the ideas in this paper assist software testers in their journey of self-reflection.

References

Austin, Rob and Lee Devin. 2003. *Artful Making: What Managers Need to Know About How Artists Work.* Financial Times.

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

Copeland, Lee. 2004. *A Practitioner's Guide to Software Test Design*. Artech House.

Gottman, John. 2010. *Relationship Repair that Works.* The Gottman Institute. http://www.youtube.com/watch?v=SqPvgDYmJnY

Patterson, Kerry and others. 2002. *Crucial Conversations: Tools for Talking When Stakes Are High.* McGraw-Hill.

Stahl, Bob and Elisha Goldstein. 2010. *A Mindfulness-Based Stress Reduction Workbook.* New Harbinger Publications.

Sutton, Robert I. 2010. *The No Asshole Rule: Building a Civilized Workplace and Surviving One That Isn't*. Business Plus.

Sutton, Robert I. 2010. *Good Boss, Bad Boss: How to Be the Best... and Learn from the Worst*. Business Plus.

Whittaker, James. 2002. *How to Break Software: A Practical Guide to Testing.* Addison-Wesley.

Building a Culture Around Exceptional Quality

Tracy Beeson
tbeeson@menloinnovations.com

Abstract

Have you ever said to yourself "I want unhappy people working on my team?" Of course not, but why?

Have you ever said to yourself "I want someone on my team who is not invested in the work we do?" Of course not, but why?

Certainly an unhappy person can work hard. Just because a person is not as invested in the project doesn't mean they can't add value. Yet despite the value, it somehow leaves us with an uneasy feeling. Why?

A person who is neither happy nor invested, is a person unable to deliver quality work at a long term sustainable pace.

This experience report is about how our team built a culture centered around delivering exceptional quality by fostering a culture of trust, respect, and joy in the workplace. It demonstrates how our team has incorporated the teachings of Dr. Deming, Patrick Lencioni's "Five Dysfunctions of a Team", Peter Senge's "The Fifth Discipline: The Art and Practice of the Learning", and others into everything that we do because we believe that the culture of an organization can have a direct affect on the quality of its deliverables.

Biography

Tracy Beeson is a senior quality advocate at Menlo Innovations in Ann Arbor Michigan. Over the past ten years Tracy has worked in the quality assurance field, the last five of which helping to build and integrate the quality assurance process into the agile process at Menlo. She has written and taught Menlo's "Building Agile Quality" class and presented at several conferences including the Agile 2008 Conference in Toronto and the PMI Global Congress 2009 in Orlando, FL.

Tracy has a bachelor of arts degree from Middlebury College in Middlebury, Vermont where she majored in Computer Science and earned her secondary teaching license in Math and Computer Science.

1. Introduction

What does it mean to be happy at work? Is that even possible? Should it be a goal? I once had a boss who thought if his employees showed signs of being happy (e.g. smiling, laughing, etc.), they weren't working hard enough. After all, he believed work should be serious, not fun.

Wrong. If work doesn't tap into your inherent joy of learning, you aren't doing it right. If you aren't happy at work, you probably aren't adding as much value as you could. That's not to say that you are doing bad work but it could mean the difference between creating something that is simply "good enough" and something that is *exceptional*.

Consider what this might mean for your company. What might be accomplished if everyone you work with ceased being content with "good work" and pushed themselves and others around them to do *exceptional* work?

This report explores why teams often fail to be exceptional, what is necessary to build an exceptional team, and why exceptional quality is so important. Using experiences from various teams, the teachings of Dr. Deming, Patrick Lencioni, and Peter Senge show us how to steer clear of the dysfunctional and build a high functioning team able to produce *exceptional* quality products.

2. Problem

I once worked on a team as a quality assurance specialist where the process went as follows: After months of meetings where we listened to everyone go around the room and explain why they were unable to get their part of the project done (usually because someone else hadn't done their part), the developers would finally deliver their code to QA weeks past the development deadline. QA was then expected to work a miracle and get the application tested in a day or two rather than a week or two. Since there was no such thing as integration for this team, it was QA's job to take all the code, put it together, and see if it would work. Inevitably, it would not.

At this point I would usually go hunting. I would walk up to the fourth floor from my basement desk and ask the web developer why it doesn't work. He would then describe to me that the problem lies with the back-end. I would then march over to the back-end developer and ask what the problem is and they would explain it was the web developer's fault. Eventually I would annoy one of the two enough for someone to take the five minutes to fix the problem. I would then march back down to my basement desk and try again until I find the next road-block and the process would start all over. It was exceedingly rare for this team to be able to release a reliably working product, much less a quality one. It was more often the case that, within the first day of release, a severe bug would be found and an emergency patch would need to be released shortly there-after.

In this particular environment, the team was used to getting blamed for its failures. In fact, managers made it clear that mistakes would not be tolerated, thus, no one wanted to admit mistakes or failure. This fear among the team fostered distrust and a CYA attitude, putting a choke hold on team communication. Problems were hidden rather than discovered and solved. Too much time and energy was spent fixing the surface level problems rather than discovering the real ones lying beneath. There was no ability to collaborate much less allow for creativity.

All too often this example resonates with people who have worked on teams that have suffered a similar pain. The culture I have described is a culture of 'me.' It is one of heroes and constant fire fighting, where the team spends all of its time fighting to build a product that simply works rather than building one of quality. Exceptional quality cannot be obtained if the team is unable to communicate, collaborate, and allow for mistakes that lead to creative solutions.

3. Exceptional Teams

No individual should have to carry the burden of being exceptional all the time. It is not sustainable and will eventually cause him to be unhappy.

Fortunately, if we build a culture of 'we,' we build a team. A team can produce exceptional work even while its individuals fluctuate between good and exceptional. In a high functioning team, concepts are first discussed with several members, but consensus is not the goal. Instead, each team member is given the chance to voice their opinions without fear of repercussions from others. With all concerns and thoughts heard, the team is able to move forward with an approach, comfortable that new information may re-open the discussion. Only then are team members confident enough to present decisions to others as "We believe…" rather than "I believe…"

In kindergarten, we teach our children that the boring and somewhat daunting task of cleaning up a messy room can be made fun by incorporating a bit of teamwork. It might even get done faster. We help each other figure out where each item belongs, learning as we go. Before we know it the task is complete. In the end, each child feels a sense of accomplishment and satisfaction that they helped get the job done.

Likewise, have you ever noticed that when you exercise with a team you push yourself harder than you would alone? The workout is significantly more effective and yet you barely noticed because you were having fun challenging yourself and your teammates. You worked hard but it was the "good kind" of hard. It felt satisfying. The team pushed you and you pushed them and together you accomplished more than you thought you would.

Dr. Deming once said:

> "Our prevailing system of management has destroyed our people. People are born with intrinsic motivation, self-respect, dignity, curiosity to learn, joy in learning. The forces of destruction begin with toddlers- a prize for the best Halloween costume, grades in school, gold stars- and on up through the university. On the job, people, teams, and divisions are ranked, reward for the top, punishment for the bottom. Management by Objectives, quotas, incentive pay, business plans, put together separately, division by division, cause further loss, unknown and unknowable." (Senge, p.xii)

Too many organizations foster a culture of 'me.' To be the best at what you do means to be better than everyone else around you. To prove you are the best is to say to your boss "I did that." To get to the top means someone else didn't. To stay at the top means to continually prove you belong there above anyone else. An organization that fosters a culture of 'me' can never build an *exceptional* team. A team requires a common goal. An exceptional team requires all members to put aside the personal in order to achieve the common goal. In a culture of 'me' team members act on their own personal goals above all else because the incentive for showing their values above all others is too high, even when they might think they are acting for the greater good of the team.

Who do you want at the top of your organization? Someone taught to constantly defend his right to be there or someone who works to bring others up with them?

Beware of the words "I" or "me." They spread like a plague upon a team and will undermine its ability to be *exceptional*.

To build an exceptional team, we must first understand the forces that work against it. Patrick Lencioni describes the core dysfunctions that will render a team ineffective:

(Lencioni, p188)

Trust rests at the bottom and acts as the foundation of the team. Without trust among team members, a team is incapable of effectively solving problems. Lack of trust leaves the team vulnerable to suffering from the other four dysfunctions and ultimately from an inattention to results. Team members are unable to put aside their personal agendas and work towards the good of the team. Their behavior reflects that of one on the defense. They are in survival mode; desperate to show they add value above the others. They are functioning in a culture of 'me.'

One such example can be taken from a group of international interns I once worked with who were asked to build an iPad® application. While the team was made up of competent individuals in varying stages of their schooling, the team suffered from a lack of trust among its members. As a result, the team members each aimed towards pushing their own personal solutions rather than working together. Despite the team's efforts to come together and decide upon a solution, each task was tackled individually. When the parts came together, they did not make a whole and a total solution was not accomplished. Upon showing the product to the client, it became clear that the solution was not satisfactory and the team resorted to bickering over who's solution would have been better in an effort to one-up each other. Both client and team were frustrated and the project came to a standstill.

Lack of trust will make moving a project forward feel like pushing a boulder uphill by hand. A foundation of trust is a necessary ingredient when making an *exceptional team* (the basis for building a strong team and avoiding the five dysfunctions), along with a shared joy in learning, and a desire to find problems in need of solutions (as opposed to solutions in search of a problem). With those in hand, an *exceptional* team will not only move mountains, they will make doing so look effortless.

3.1 Building Trust

Tom and Birgit Hanson provide a model for acting "*in* integrity" thereby building trust and accountability with the people with whom you interact. They define integrity as honoring your word and doing what you said you would do. By not doing so, you pay the price twofold: First, what you said you would get done didn't get done. Second, the person you promised will be less likely to trust that your next promise will be honored, causing any future interactions to be strained. (Hanson, p 48)

It sounds so simple. You might even think you are already "*in* integrity." How often are you late to a meeting without letting others know? How many times have you blown an estimate and avoided telling your manager? How many times have you promised to get that report done only to not have it ready when asked? Will your co-workers trust those promises the next time? The model shows how easy it is to fall out of integrity and opens our eyes to the effect we, as individuals, have on a team.

If trust is the building block towards avoiding a dysfunctional team, then we must consider our individual affect on trust within the team. We must act with integrity and earn the respect of our teammates.

In the example of the international interns, we applied two strategies for building trust. The first was to have a very open and constructive conversation about the behaviors each intern was exhibiting and to discuss why those behaviors were working against the success of the team. It was not an easy conversation, but a necessary one. The second was to bring other members of the greater team in to model a collaborative approach to solving problems, including acting with integrity. Not all of the interns were able to internalize the feedback they were given and reflect on the consequences of their actions. Collaborative thinking is not for everyone. It is important to respect that people are different and excel in different environments.

Trust is not something we cannot force upon our teammates. It must be earned. Earning someone's trust requires looking inward at why you might have lost their trust in the first place. This can be an unpleasant exercise that should not be taken lightly. Those who are able and willing to partake in the exercise, no matter how unpleasant, will make excellent members of an *exceptional* team.

To build an exceptional team, we must also foster learning. "Organizations learn only through individuals who learn. Individual learning does not guarantee organizational learning. But without it no organizational learning occurs." (Senge, p.129) An *exceptional* team is a learning team.

3.2 Joy in Learning

As Deming points out, we are all born with an inherent joy in learning. Yet along the way the prevailing system of management builds a culture of "me" thereby suppressing our natural eagerness to learn. We become stagnant in our roles, "putting in our time" until we can be in charge.

An acquaintance of mine worked for a small company of designers. Management had set forth the goal of moving into a new part of the field. It just so happened that this new sector was his specialty. Being the avid reader that he is, he began forwarding interesting articles on the topic to the team in an effort to share the knowledge. His manager pulled him aside and requested that he no longer send such articles, noting that doing so was a bother to his co-workers. His immediate response to the request was to stop sending articles and stop investing his time and energy into the company. Needless to say, as soon as he was able, he found a new job.

On the other hand, my current place of work, Menlo Innovations LLC, encourages all members of the team to not only learn but to be aware of the many different learning styles other team members may have. This allows us to teach each other more effectively. Whether you are new to the job or a seasoned veteran, young or old, if you are not learning, you are teaching. Sharing of knowledge and appreciating the different perspective people bring towards understanding the same topics enables the team to gain more clarity on the problems we need to solve. We rely on the constant cycle of learning, making mistakes, gaining perspective, understanding, and re-learning to push a project forward. It is a vital part of our process and has been crucial in gaining widespread adoption for the products we build.

To work is to learn.

If we are not learning, we are not working. To be clear, I am not referring to the kind of learning that goes on in most schools. It is the difference between learning a skill and the art of learning itself. Peter Senge teaches us that learning is an art and we must regard it as a continual process, ever changing our perspective on the world around us.

For a team to avoid becoming stagnant it must be composed of individuals that share in the joy of learning and see value in mastering the art of it.

3.3 A Problem in Need of a Solution

At Menlo Innovations LLC we give tours of our factory daily. Our CEO, Rich Sheridan, proudly and patiently walks inquisitive business folk around giving just a taste of what it is that we do and how we do it. Some are eager to soak in every detail and some are not. Those who are not, usually look at the photocopied note cards we methodically post on the walls in swim lanes with thinly veiled skepticism. Upon occasion they are bold enough to attempt to poke holes in the value of posting these project boards all over the walls in the factory only to be thwarted by our team's openly honest responses. About ten times a year one such question is "Why don't you put the story cards in a software tracking program so you can track them electronically?" To this we emphatically reply "because no one would look at it."

A solution in search of a problem is one that solves a problem that, to the users, never existed. Sure, to the average passerby it looks a bit crazy to manage multi-million dollar projects using paper, note cards, colored dots, and thumb tacks but to us, it makes total sense.

It is this key ingredient that requires the team to broaden the definition of quality. When we define quality as "the product performs to the specification" we lose sight of what problem we were trying to solve for the user in the first place and neglect our responsibility as team members to build a product that meets the user's needs.

In order to keep a project focused on meeting the quality needs of the user it is the team's collective responsibility to beg three questions:

"What problem are we trying to solve?" Since it is quite common that people describe a problem by its solution, it is important to constantly remind ourselves of the problem itself. This is why, at Menlo Innovations LLC, one of the first things we do when we begin a project is to help our clients articulate the problem they believe their product will solve.

"Whose problem is it?" A solution for me may not work as a solution for you. There is no such thing as one size fits all. However we can strive for one size fits most by strategically choosing whose problem we solve.

"Will that solution work for the user?" We must shed our preconceived notions of what might work and remind ourselves that the solution does not need to make sense to me. It needs to make sense to the user.

A high functioning team that shares in the joy of learning and is centered on a problem to solve *is* an *exceptional* team.

4. Producing Exceptional Quality

In one of my past jobs as a quality assurance professional, I was quietly working at my desk in the basement of the building when my phone rang. It was my friend, a developer from the fourth floor. She asked if it would be okay for her to get a copy of the document QA keeps that lists how certain functions work in the application. She wanted to update it with the information the developers were keeping. It seemed like a perfectly reasonable request to me. This is when I came to find out that she had made the same request from one of my QA team members only to be told no, she could not have the document. After opening the discussion with my QA team, it was clear that my QA companion really just didn't want to share. This being just one example of dysfunction among the team, it was no wonder why the software we worked on was plagued with bugs. After a bit of debate it was determined that sharing the document would benefit us all (QA and developers). As for my QA companion, he continued to build animosity between himself and the developers ultimately hurting the projects he worked on and making his job more difficult.

Only an *exceptional* team can produce exceptional quality. Consider the iPhone®. It epitomizes exceptional quality. Sure they have had a few technological glitches along the way but they managed to roll out a world changing technology and did it without leaving one stone unturned. From the buyer's experience, to the packaging, to the phone itself, to the app store, to development of new applications, they thought of it all. From what you always wished for in a phone to what you couldn't even imagine was possible, they thought of it and implemented it. Quality is not just about whether your phone works. It is about the total experience. You can't accomplish this kind of quality with a dysfunctional team.

Sure it is possible to have success without quality. It is not possible, however, to sustain the success without it. It wouldn't take long for each of us to think of a wildly successful software application that would not pass our test of quality as users. Do you think that application will exist 10 years from now? Will the iPhone®? No matter what industry, to survive in today's fast paced world of technology widespread adoption is not just a goal. It is essential.

5. Conclusion

As Dr. Deming teaches us, you cannot build quality after the fact. You must build a system that fosters it. To build a culture around exceptional quality is to change our system of thought around what quality means and how we achieve it. Users expect more than a product to simply work. There is no longer room for "good enough."

Build an *exceptional* team, and see how effortless it can be to build *exceptional* quality products.

References

Hanson, Tom, and Birgit Zacher Hanson. *Who Will Do What by When? How to Improve Performance, Accountability and Trust with Integrity*. Power Publications, Inc., 2007.

Lencioni, Patrick. *The Five Dysfunctions of a Team A Leadership Fable.* San Francisco, CA: Jossey-Bass, 2002.

Senge, Peter M. *The Fifth Discipline The Art & Practice of the Learning Organization Revised Edition.* Doubleday, 2006.

QUnit JavaScript Testing in the Enterprise

C. David Lee
david.lee@sftsrc.com

Abstract

Beyond the days when JavaScript changed minor facets of a web page, today's JavaScript creates full featured web applications. Technologies such as JSON, JQuery and AJAX move the playing field from server side to client-side with much of this functionality now delivered via JavaScript.

Programmatically unit testing this client-side code gives us the assurance that it will not break. The challenge is to implement JavaScript unit testing while keeping it relevant when server-side, client-side or HTML changes are made, and while keeping test failures visible.

What does this mean for quality? If an organization has a unit testing strategy they will detect errors quicker and easier than organizations that do not. When JavaScript unit tests are written well with code coverage then changes will no longer require hours of manual testing. Removing even a small portion of manual testing can save time and money, giving a QA organization the ability to focus on what's new rather than rehashing old tests.

SoftSource consulting has thought through how to implement JavaScript unit testing in a way that saves time, focusing on some of the drawbacks and positives of various approaches, and using custom development with the QUnit testing library.

Biography

David Lee is a Senior Software Developer at SoftSource Consulting (www.sftsrc.com), a consulting firm with a proven track record of helping companies engineer custom software solutions on the Microsoft platform. David has been developing and architecting software solutions for the past 13 years. Most of his experience has been in designing and maintaining enterprise portal applications in the financial/professional services industry. David graduated from Oregon State University with a BS in Liberal Studies.

1. Introduction – Why JavaScript Unit Testing?

Have you ever wondered how to make unit testing strategies apply to JavaScript? Technologies such as JSON, JQuery and AJAX provide the functionality via JavaScript that was once provided server-side testable code. This code is complex and rich. Through testing small "units" of functionality in an automated way this complicated JavaScript code can be made more solid, less breakable and with good test coverage make refactoring easy and inexpensive.

Programmatically unit testing client-side code gives us assurance that it will not break when changes are introduced. Those who work in an enterprise web environment know things change frequently and they may not always be aware of how or why the change took place.

In an enterprise, code originates from multiple layers (server-side, client-side), mixes of technologies (3rd party, legacy code, brand new controls) and the HTML itself can be dynamic changing once or twice on every layer. To survive all of this complexity a need for automated testing is in order. It keeps our code stable during maintenance, clear when investigated and saves time in rehashing old stuff. As we delve more and more into JavaScript, the stage for JavaScript unit testing clearly is set.

This paper has two sections. One portion of the paper will focus on some of the concepts of client-side testing and the other portion will show examples in how JavaScript testing is implemented.

1.1 Requirements for this paper… Assumptions regarding the reader

This paper is focused on the enterprise environment and primarily for those who have unit testing already well in hand. However, anyone who has a web site, who writes JavaScript code, or who has to test JavaScript web applications should strongly consider investigating JavaScript unit testing as described here.

The assumptions of this paper are that the reader first-off has a basic understanding of HTML and JavaScript. It will discuss JavasScript and HTML fluidly. An understanding of Unit testing processes, JQuery selectors [(2)] and the concepts of server-side versus client-side programming are helpful. Some Microsoft technologies are mentioned in this paper and one of example is written in Microsoft ASP.Net/C#.

1.2 In the Enterprise, some thoughts

How tests are run is often just as important as running the tests themselves. Preparing a proper unit testing strategy for your organization is only as complicated as your needs and requirements. It could be simple or it could be quite complicated. With that said, however, keeping things simple at first is usually the best strategy, then building up as the needs arises.

Here are some additional thoughts:

- piggyback upon your existing unit testing strategies.
- leverage your test or your development environment
- make sure your build process respects these tests, not placing them into production
- provide for reporting

One key to success is by adding visibility to JavaScript testing by making your test failure or success notifications visible as needed. These should not go on hidden, but rather be raised up to those who should take action.

2 Testing strategies: to HTML or not to HTML…

Will you add HTML into your JavaScript unit tests? This is not quite the simple question that some would assume it is. Some would easily say, "No way!", but HTML is so closely tied to JavaScript functionality that the waters soon become muddied and HTML may become a necessity.

2.1 A natural division between code and the UI

Experts agree that unit testing is best applied to the code that deals with logic, not to code that shows the User Interface itself. For example if you have a calculator, you generally do not try to unit test the size and position of the buttons. This should be left up to a graphic designer and made changeable. You should also not unit test the color of the display or the font size of the numbers displayed for the same reasons. These things should be left up to a design team and not fixed in code by programmers.

However, this is what HTML is. HTML is the size of the buttons, the number of buttons, the height and width of areas on the screen. The clear division between testable code and other code should say, naturally, that HTML is not testable because it ***is*** the user interface.

The problems come where there is major functionality implemented in JavaScript and where JavaScript has to rely on the fact that buttons, accordion controls, IFRAME objects and all manner of client-side HTML objects exist.

2.2 What we knew before we got started

When we looked at implementing JavaScript unit testing there were some different testing strategies considered.

Possibilities were:

1. separating JavaScript class libraries out so that they could exist completely independently of web pages
2. creating integrated testing where JavaScript is tested in the context of each page
3. copying HTML into our tests
4. creating static, never changing, test HTML

However, when we realized what we already knew, this simplified our lives. We already knew…

- **JavaScript is very close to the User Interface…** in fact it runs in the user interface. It manipulates HTML, reads from FORM fields, and is often designed to manipulate the display of web pages. Most JavaScript does something with HTML.

- **JQuery brings JavaScript even closer to the HTML…** with selectors. JQuery selectors read HTML elements from the HTML DOM. Most actions taken against JQuery is to collections of HTML objects found through selectors.

- **HTML is fluid…** and changeable. Most commonly used server-side controls create HTML and JavaScript. They might change structure right out from underneath us as programmers.

- **Designers and developers…** both change HTML markup**.** Developers may write JavaScript, but designers change the layout of HTML for design purposes. Look and functionality do not need to oppose each other, but often times the very structure of the HTML may affect hidden JavaScript

relying on specific formatting. This might lead to defects only visible by clicking through a web page.

2.2.1 An example of why it may be difficult to separate HTML from JavaScript

Here's an example:
To perform a simple task in JavaScript many use JQuery,… because it is easy. JQuery selectors, however, are really just "HTML queries" relying on HTML tags, class names and HTML ID's. The HTML itself is important to many JQuery operations.

Example: If you wanted to clean out the child tags under a certain set of parent HTML elements you could quite possibly do something like this:

```
$(".className > div").empty();
```

Basically, this says, "Give me every "<*div*>" tag that has a parent tag with a class assigned to it named "*className*"." Ouch!!! No separation here. In order to test this code we would have to have a small segment of HTML with a bunch of tags assigned the "*className*" class and that have "<*div*>" tags underneath them.

2.3 JavaScript tests may need HTML…

So, what does that mean for JavaScript testing? It means there are some underlying problems in the separation of testable code and the HTML that it changes. The HTML and the JavaScript it modifies are so closely tied together that at times JavaScript code will not work without HTML.

Our conclusion was that there is enough JavaScript that requires HTML that without HTML many of our JavaScripts will not be unit tested at all. If we were to test JavaScript without HTML then much of our code would be untested. Therefore our challenge was just how to put HTML into our JavaScript tests while honoring the fact that HTML is changeable, manipulated on a page in any number of ways. In other words we were asking, "How can we bring HTML from these pages directly into the tests we were running?"

2.4 Creating HTML ?

Now that we decided to use HTML we recognized that we had to do it "the right way!" This meant avoiding the outright creation of HTML for the tests. Instead we opted for using HTML directly from the page.

The reason we avoided creating HTML is illustrated in a simple example:

Suppose we copied a simple accordion control out of our web pages and pasted it into a test. Our tests were great and ran efficiently, always passing. Sometime later, when a programmer or designer came along to enhance the page (perhaps changing tags to tags) the copy of the accordion in the test will not get updated! This change introduced a problem that would go completely unnoticed by the unit tests because the tests were not aware of the problem!

Static HTML can become old and irrelevant.

To all concerned, the updates were done and still the tests were passing, but the obvious problem was that we created static HTML in the first place. If you've ever studied for an exam and found out later that you studied for the wrong one… you now know why copying and pasting HTML into a test is a bad thing! Testing an old and stale copy is not the same as testing the real thing!

2.5 Directly testing the page?

The logical path we were following was taking us to someplace that was fairly unique.

Nowhere else in web development are we asked to:

1. create unit tests,
2. however, include a snippit of the actual UI of the page in those tests

This was now different than anyone had anticipated. The problem is that we now lived under one more constraint if we were to include actual HTML markup in our tests.

> We had to not impact the current web pages themselves!
> We had to keep the behavior, look and feel exactly the same.

We knew that our tests should have little to no impact on the page against which they ran.

For example, if an accordion control was to have a certain message when opened. Any tests that ran should not modify that message or change the message formatting in any way. So, our challenge was to create a process that used real HTML (from the page), but did not change that HTML on the page. It should not impact the page or cause undesired behavior.

What we chose to do was place Unit tests in a hidden IFRAME, and to put a small status dialog on each page that lets us know the state of each test. (See section 5)

3 What is QUnit?

In order to run tests first we need a framework in which to run them.

What is QUnit? To quote http://docs.jquery.com/Qunit,…

> "QUnit is a powerful, easy-to-use, JavaScript test suite. It's used by the jQuery project to test its code and plugins… [It] is capable of testing any generic JavaScript code."

To understand QUnit please look at http://docs.jquery.com/Qunit for a full API reference, but to get a quick preview of its capabilities…

3.1 QUnit Setup

- module(name, lifecycle)
  Separate tests into modules.

- test(name, expected, test)
  Add a test to run.

- asyncTest(name, expected, test)
  Add an asynchronous test to run. The test must include a call to start().

- expect(amount)
  Specify how many assertions are expected to run within a test.

3.2 QUnit Test Assertions

QUnit has several built-in functions that allow you to assert when something is true or false, therefore making your tests pass or fail.

Assertions possible with QUnit:

- ok(state, message)
  A boolean assertion, equivalent to JUnit's assertTrue. Passes if the first argument is true.

- equal(actual, expected, message)
  A comparison assertion, equivalent to JUnit's assertEquals.

- notEqual(actual, expected, message)
  A comparison assertion, equivalent to JUnit's assertNotEquals.

- deepEqual(actual, expected, message)
  A deep recursive comparison assertion, working on primitive types, arrays and objects.

- notDeepEqual(actual, expected, message)
  A deep recursive comparison assertion, working on primitive types, arrays and objects, with the result inverted, passing when some property isn't equal.

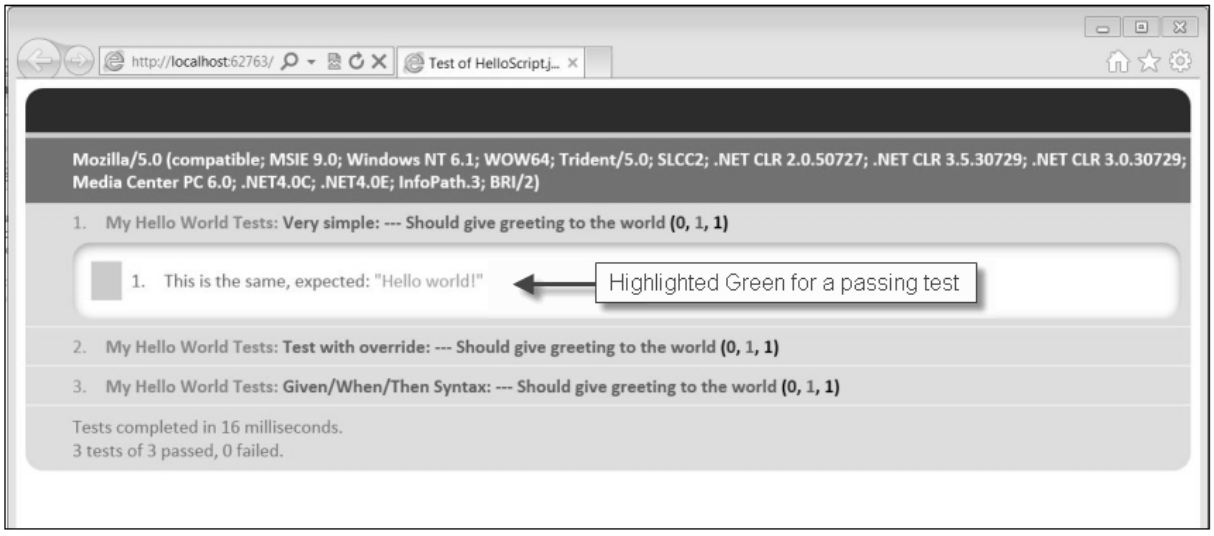
- strictEqual(actual, expected, message)
  A stricter comparison assertion then equal.

- notStrictEqual(actual, expected, message)
  A stricter comparison assertion then notEqual.

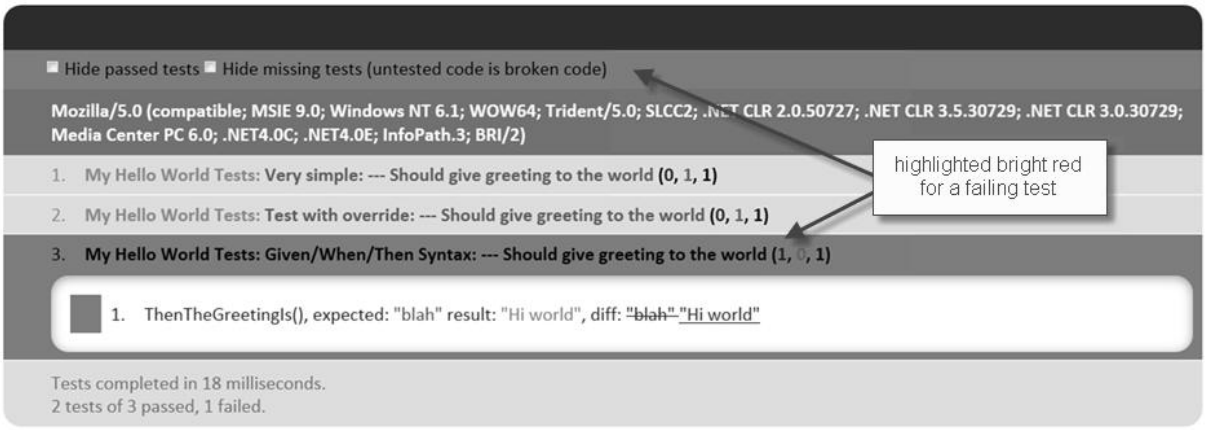
- raises(block, expected, message)
  Assertion to test if a callback throws an exception when run.

3.3 The Test Runner

3.3.1 Image of a passing test



3.3.2 Image of a failing test



4 Some examples of testing with QUnit

4.1 A Basic Test

This is a very simple example of testing with QUnit. In this example the test simply checks the output and asserts true if it is what is expected.

4.1.1 Web Page

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title>Sample Production page with external JavaScript.</title>
    <script src="HelloScript.js" type="text/javascript" language="javascript"></script>
</head>
<body>
<h2>Sample Production page with external JavaScript.</h2>

<div style="font-style:italic">
<script language="javascript" type="text/javascript">
    var greeting = GetHello();
    document.write(greeting);
</script>
</div>

<div style="padding-top:5px;">
    <div><a href="TestPage.htm">Click here</a> to view the tests for this page</div>
</div>

</body>
</html>
```

4.1.2 Script to be tested: HelloWorld.js

```
var helloWorld = "Hello world!";

function GetHello() {
    return helloWorld;
}
```

4.1.3 Test Runner Page

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
```

```
<head>
    <title>Test of HelloScript.js JavaScript file</title>
    <link type="text/css" href="../Common/qunit/qunit.css" rel="stylesheet" />

    <script type="text/javascript" language="javascript"
    src="../Common/scripts/jquery-1.4.1.js"></script>

    <script type="text/javascript" language="javascript"
    src="../Common/qunit/qunit.js"></script>

    <script src="HelloScript.js" type="text/javascript" language="javascript"></script>

</head>
<body>
<script type="text/javascript" language="javascript" >

    module("My Hello World Tests");

    test("Very simple:  ---  Should give greeting to the world", function () {

        var helloTest = GetHello()
        same("Hello world!", helloTest, 'This is the same');

    });

</script>
    <div>
                    <h1 id="qunit-header"></h1>
                    <h2 id="qunit-banner"></h2>
                    <div id="qunit-testrunner-toolbar"></div>
                    <h2 id="qunit-userAgent"></h2>
                    <ol id="qunit-tests"></ol>
                    <div id="qunit-fixture">
                           test markup, will be hidden
                           <div id="testHtml" style="display:none;" ></div>
                    </div>
    </div>
</body>
</html>
```

4.1.4 Output of test results

4.2 A test the override of a variable

To build upon the 1st example… sometimes it is appropriate to "mock" or "spoof" a variable or a class for testing purposes. This is very easy in JavaScript as seen in this example.

4.2.1 Unit test script to be placed in the test runner page

```
module("My Hello World Tests");

test("Test with override:  ---  Should give greeting to the world", function () {

    helloWorld = "Hi world!"
    var helloTest = GetHello()
    same("Hi world!", helloTest, 'This is the same');

});
```

4.2.2 Output of test results

4.3 Testing using a typical "Given/When/Then" testing syntax

Also building upon the previous examples… we can use good formatting. Today's testing, in many cases, uses a syntax called "Given/When/Then". This syntax is just as easy in JavaScript as it is in C# or any other server-side code.

4.3.1 Test Runner Page

```
test("Given/When/Then Syntax:  ---  Should give greeting to the world", function () {

    GivenTheGreetingIs('Hi world');
    WhenTheGreetingIsRequested();
    ThenTheGreetingIs('Hi world');

});


var _returnedGreeting;
```

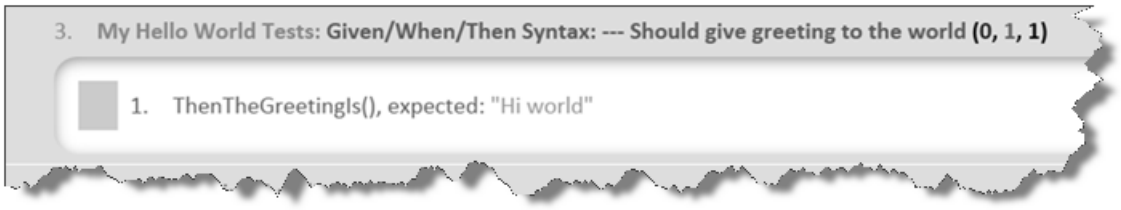
```
function Setup() {
}

function GivenTheGreetingIs(greeting) {
    helloWorld = greeting;
}

function WhenTheGreetingIsRequested() {
    _returnedGreeting = GetHello();
}

function ThenTheGreetingIs(greeting) {
    same(_returnedGreeting, greeting, "ThenTheGreetingIs()");
}
```

4.3.2 Output of test results



5 Implementation of JavaScript testing using our strategy

Our compromise, as mentioned in the end of section 2, was to include fresh HTML in our tests that is quickly copied out of the original page at the start of each test run. It runs tests after the page load has completed, placing HTML into a hidden IFRAME which runs our tests. Following test completion the status is reported back to a dialog.

Written in C# and ASP.Net, this example includes all of the strategies mentioned earlier.

Our solution strategy included 4 parts:

1. **Start from a codebase used site-wide:** To make the tests site-wide we wired our tests into a site-wide base class. This code verifies that it is only run in the development environment.

2. **Automatic wiring:** The test runner automatically shows up when a test file is put into a directory following the correct naming convention.

3. **Minimal footprint:** The test runner has a very low impact to the original page. It creates and destroys a *hidden* IFRAME that runs the tests. Then the results of the test are displayed in a tiny box in the upper left-hand corner.

4. **Easy to create tests:** To create a test the developer must simply create test runner page with the correct naming convention, using the provided test-runner master page.

5.1 The initial setup

A developer will need to take the sample code provided to integrate it into your site.

- We used a base class (*in the provided code samples it is a master page*) that includes both the JQuery script library and includes this code added to the OnPreRender event.
- Our production pages inherit from this base class (*again, in the test samples we are using a master page, but a similar concept*).
- Every path must be correct and changed to accommodate your site.

5.1.1 Example: Passing test

The result of the sample test is a small dialog showing success or failure. This shows at the top left-hand corner. By clicking "(show)" the testing dialog shows the QUnit test runner in an IFRAME.

5.1.2 Example: Failing test

When even one test is failing the test dialog shows as red. By clicking "(show)" you can also see failing tests.

5.2 The "using" namespace

In our example we created a JavaScript library for including things into our tests. This library is named "using." It load scripts HTML into a test automatically, based on the JQuery selector specified..

5.3 The "using" Class

- using.SelectedHtml(jquerySelector)
  Uses JQuery selectors to find HTML *on the parent page* and load it into the test page.

5.4 To implement a test using this strategy

Take a look at the sample code provided. Once this sample code is integrated into your site then the process of creating a test is fairly simple:

1.) Create the correct directory path for your tests
2.) Create an ASP.Net .aspx page in this new directory (following a naming convention)
3.) place your tests in the .aspx page.

6 Some Additional Testing Ideas

Here are some JavaScript testing ideas, thoughts on how you could use QUnit. These don't have corresponding examples in the code provided, but they are submitted as some food for thought.

1. **Make sure your web services are called** – By mocking up the return from JSON calls you could validate that your JSON return methods are coded correctly.

```
// loads the parent page's scripts into memory
using.ScriptsFromParentPage();

// a QUnit command to create a testing module
module("HealthConcierge", { setup: Setup });

test("Should Verify Autocomplete Webservice was called", function () {

    GivenAFakeSearchPrefix('fakeSearchPrefix');
    WhenAutoCompleteIsInvoked();
    ThenAutoCompleteServiceWasCalledWith('fakeSearchPrefix');
    ThenAutoCompleteServiceReturnedResults(_mockResults);

});


function GivenAFakeSearchPrefix(searchPrefix) {
    _searchPrefix.term = searchPrefix;
}

function WhenAutoCompleteIsInvoked() {
    _webService.CallService(_searchPrefix, TestResultsFunction);
    _mockedWebServiceOptions.success({
        d: _mockResults
    });
}

function ThenAutoCompleteServiceWasCalledWith(searchPrefix) {
            same(_mockedWebServiceOptions.data, "{\"searchText\":\"" +
searchPrefix + "\"" + ",\"numberOfTermsToReturn\":\"" + "20" + "\"}",
"ThenAutoCompleteServiceWasCalledWith('" + searchPrefix + "')");
}

function ThenAutoCompleteServiceReturnedResults(results) {
    same(_resultFromWebServiceCall, results,
"ThenAutoCompleteServiceReturnedResults('" + results + "')");
}



function Setup() {
    jQuery.ajax = function (param) {
        _mockedWebServiceOptions = param;
    }

    _webService = new WebService();
}
```

```
var _webService;
var _mockedWebServiceOptions = null;
function _searchPrefix() { term: null };
var _resultFromWebServiceCall;
var _mockResults = 'foo';

var TestResultsFunction = function (message) { _resultFromWebServiceCall =
message; }
```

2. **Validate the return from JavaScript functions –** QUnit is very similar to NUnit, so most of the assertions you are used to making can be made in JavaScript too.

```
// loads the parent page's scripts into memory
using.ScriptsFromParentPage();

module("PublicPage");

test("Is it Four?", function () {
        WhenTheCountIsCalled();
        ThenTheResultIsFour();
});

function WhenTheCountIsCalled() {
        countResult = new CountResults().Count();
}

function ThenTheResultIsFour() {
        same(countResult, 4, "ThenTheResultIsFour");
}


var countResult = 0;
```

3. **Validate JavaScript against Mocked objects –** JavaScript is very good at mocking objects because nothing is strongly typed.

```
using.Script("/HealthConcierge/ConditionResults/TreatmentCosts.js");

module("TreatmentCosts");

test("Should Call Counts", function () {
        WhenTreatmentCostsResultsAreRendered();
        ThenCountsAreRendered();
        ThenPresentIsCalled();
});
```

```
function WhenTreatmentCostsResultsAreRendered() {
        var treatmentCost = new ConditionResults.TreatmentCosts();
        treatmentCost.presenter = new MockPresenter();
        treatmentCost.Present(null);
}

function MockShowCount(resultLocationElement, results) { countMockCalled =
true; }
function MockPresent(treatmentCosts, template, resultLocationElement) {
presentMockCalled = true; }

function MockPresenter() { }
MockPresenter.prototype.ShowCount = MockShowCount
MockPresenter.prototype.Present = MockPresent;

function ThenPresentIsCalled() {
        ok(presentMockCalled, "ThenPresentIsCalled");
}

function ThenCountsAreRendered() {
        ok(countMockCalled, "ThenCountsAreRendered");
}

var presentMockCalled = false;
var countMockCalled = false;
```

4. **Validate objects in the page** – For example, if your scripts rely on several HTML objects in your page you could test to make sure these objects exist

```
// a QUnit command to create a testing module
module("HealthConcierge", { setup: Setup });


test("Does this HTML exist?", function () {

   DoesThisExist("body"); // the body tag

   DoesThisExist("#testHtml"); // a tag with a “testHtml” id

   DoesThisExist("#results > div"); // a tag with a “results” id with <div> tags
under it

});



function DoesThisExist(jQuerySelector) {
   ok($(jQuerySelector, window.parent.document).length, " DoesThisExist (" +
jQuerySelector + ")");
}
```

```
function Setup() {}
```

5. **Validate the loading of other scripts** – sometimes scripts get renamed or deleted. You could validate the objects instantiated by any other JavaScript reaching out to the parent page if desired.

```
// loads the parent page's scripts into memory
using.ScriptsFromParentPage();

// a QUnit command to create a testing module
module("HealthCareProvider", { setup: Setup });


test("Should Have ConditionResults Presenters", function () {

    ThenThisPresenterIsLoaded(parent.ConditionResults.Conditions,
"ConditionResults.Conditions");

    ThenThisPresenterIsLoaded(parent.ConditionResults.Discussions,
"ConditionResults.Discussions");
    ThenThisPresenterIsLoaded(parent.ConditionResults.DoctorQuestions,
"ConditionResults.DoctorQuestions");

    ThenThisPresenterIsLoaded(parent.ConditionResults.Drugs,
"ConditionResults.Drugs");
    ThenThisPresenterIsLoaded(parent.ConditionResults.HospitalQuality,
"ConditionResults.HospitalQuality");

    ThenThisPresenterIsLoaded(parent.ConditionResults.Provider,
"ConditionResults.Provider");
    ThenThisPresenterIsLoaded(parent.ConditionResults.TreatmentCosts,
"ConditionResults.TreatmentCosts");
});



function ThenThisPresenterIsLoaded(presenter, presenterName) {
    ok(presenter, "ThenThisPresenterIsLoaded(" + presenterName + ")");
}

function Setup() {}
```

7 Conclusion

JavaScript testing is not a necessity, but it can save you time and money as you properly plan an implement a JavaScript testing strategy. With the proper infrastructure and a few simple steps, programmers can add JavaScript tests to any page.

Definition of Terms

- HTML — HyperText Markup Language – the building blocks for web pages
- HTML DOM — the document object model of an HTML document; basically, the structure and objects within the document.
- Markup — In this paper the terms “markup” and “HTML” are used interchangeably, however, markup language itself is defined as a way to annotate text through using a specific syntax. (for example, specifying the creation of a table through using the syntax <table></table>
- JavaScript — JavaScript is a prototype-based, object-oriented scripting language that is dynamic and weakly typed.
- C# — (pronounced see sharp) is a programming language developed by Microsoft encompassing strong typing, object-oriented (class-based), and component-oriented programming.
- ASP.Net — ASP.NET is a web application framework developed and marketed by Microsoft to allow programmers to build dynamic web sites, web applications and web services.
- jQuery — jQuery is a cross-browser JavaScript library designed to simplify the client-side scripting of HTML. It was released in January 2006 and is used in over 46% of the most visited websites.
- UI — User Interface
- IFRAME — The IFRAME tag is an HTML markup designation that creates a frame. Frames allow a visual HTML Browser window to be split into segments, each of which can show a different document.

References

1. Code samples referenced here are found at: http://sftsrc.com/content/docs/PNSQC2011_QUnit.zip

2. www.jquery.com (JQuery)

3. http://api.jquery.com/category/selectors/ (JQuery Selectors)

YES! You CAN Bring Test Automation to Your Company!

Alan Ark
QA Manager
Compli
arkie@compli.com
http://www.linkedin.com/in/arkie

Abstract

No money? No support from your supervisor? No experience in test automation? No problem.

Even with these barriers to entry, you too can reap the rewards of automating test activities in your web application. Sometimes a formal framework is overkill. Shooting for the moon rarely works when there are so many unknowns. With a less than optimal implementation, bad test automation could also leave your project at risk.

This presentation will provide you with an idea framework to help your test efforts along. The key is to start small and learn as you go along. By utilizing Ruby (a free scripting language) and the Watir (Web Application Testing in Ruby) library among other tools, the hit to your budget is zero dollars. By building on cumulative successes, you can present the business case for supporting automated test efforts in your organization.

As someone who has brought in test automation to several organizations, I will share with you my keys to success, so that you may also reap the many rewards of automated testing on your projects.

Biography

Alan Ark is the QA Manager at Compli, in Portland, Oregon. Alan has gained tremendous experience working for Unircu, Switchboard.com, and Thomson Financial - First Call. Mr. Ark has previously presented 'Euro: An Automated Solution to Currency Conversion' at Quality Week '99, and 'Collaborative Quality: One Company's Recipe for Software Success' at PNSQC 2008. At Compli, he is using Ruby to solve problems both large and small. His LinkedIn profile can be viewed at http://www.linkedin.com/in/arkie.

Introduction

In the course of our day to day lives, we tend to work on the same areas again and again. In some cases, to the point of taking the exact same steps on newer iterations of the project. Isn't there a way that we can make the process a little less painful? Usually there are ways to accomplish this very idea.

The idea of test automation can be expanded to include ideas that are not test cases in themselves, but that need to be in place for running of the tests. Some examples are setup of test environments, clearing out log files, and creation of data. Creation of automated test cases is also key to helping you become more efficient in your job. The use of Watir and other Ruby libraries can help with the regression testing of various web applications. In fact, using a few other libraries on top of Ruby, you can even interact directly with databases!

We'll examine some of these concepts using my experiences with the goal being to help your company realize similar gains in efficiency as well. It is my hope that this session will be more give and take rather than a single voice speaking to a crowd. While I have worked in companies that utilize Microsoft tools, the ideas can be generalized to most other development toolsets.

The main idea of this paper is to present test automation in a different light. By showing how automation can be used not only the execution of test cases, you should be able to gain more leeway from the stakeholders in your company for more test automation activities in the future.

The section titles come from song titles by Jimmy Buffett. He happens to have a song for every occasion.

1. "If the Phone Doesn't Ring, Its Me"

The usual reasons I hear that test automation is not used in various companies.

Price
- The company does not have the budget.
- We can't afford the training.

Support from management
- The company doesn't want to buy the tools.
- My manager doesn't want to spend the time automating tests.

Experience
- We tried it once, but it failed miserably.
- We don't have people who know how to program.
- I can't do it.
- It's too hard.

2. "Bend a Little"

Pouting is not going to get you anywhere. Realize that when you give a little that maybe you'll get a little in return. You may not get all the resources that you are asking for, but by showing some success, that opportunity is closer than it was before.

Think about what you really want, and go explore some options. At first, you may have to do some of this analysis "off the books" until you have some solid gains to show. Take the time to really show how these methods can shine in your environment.

Fit the exploration into your day to day activities.
- Be flexible.
- You might have to do some of this research outside of office hours.

Recognize when things are not going as well as you had hoped.
- Prevent this activity from becoming a rabbit hole.
- If you are spending too much time on the research, maybe its not the right solution

Plan on getting interrupted
- Realize that this is not job number one for you.
- Document progress so that you can continue with the project as time permits.

3. "What If The Hokey Pokey Is All It Really Is About?"

Is it an elaborate tribal custom or something simpler? Sometimes, simpler really is better. Especially when starting with something new. Realize that all projects are unique in their own special way, but to recognize and leverage patterns when they appear.

What's the plan? People like to hear what has worked for others. This provides validation that someone else has achieved some level of success with their work. The real question that needs to be answered is "What are MY pain points?" By learning what YOUR pain points are, only then can a solution that works for your situation be crafted.

This is a software development project
- Multiple people can be working on separate issues
- Measure how much time the tasks are taking
- Get a good handle on how much time people are taking on the tasks - including yourself!
- Have a goal – What do you want to accomplish
- Have a high level plan – What steps are needed

Break out the pain points into smaller problems to solve
- Divide and conquer
- See if the problem breakdown is a well defined set of problems to solve
- If it is, it's a candidate for automation
- If it isn't, then you might have to rethink your plan of action
    - Reframe the question
    - Break the project up in a different way
    - Know when you are in over your head

See if automation can help move the testing process forward in different areas
- There are certainly cases where automation should not or cannot be used
    - OS and software installs.

Don't make things more complicated than they need to be
- Simpler really is better when you're first starting out

Example of something that is not truly "automation" but can help
- Configuration of registry settings
    - Importing them from a file works great
    - Big improvement over entering them manually
    - Able to rapidly configure multiple machines with the same configuration now

Types of things I do.

Seeding test data
- Use Ruby and the FasterCSV gem to generate comma separated value (CSV) files

Drive browsers
- Use Ruby and the Watir gem to interact with web sites

Push out web builds.
- Use command line scripts to take care of
  - Cleaning out directories
  - Pushing out the new files
  - Reregistering the libraries used by our product
  - Recycling Internet Information Service (IIS) on our servers
  - Turns a web rollout into one line at the command prompt

Create log files.
- Use Ruby and the File library to create useful log files
- Log errors
- Log status messages

Clean out old logs
- Can use Ruby and the File library
- Can use the command prompt directly

Interact with databases
- Use SQL Server Management Studio directly
  - Query Analyzer
  - Object Explorer
- Use Ruby with the following gems
  - Database Interface (DBI)
  - Open Database Connectivity (ODBC)

Interact with CSV files
- Use Ruby with FasterCSV to consume CSV files
- Use Watir to interact with the web site using the read data

Load testing
- Use OpenSta, an open source load testing tool
- Not for beginners, but listed here as a reference

Put some of the above tasks together
- Login/logout of multiple users
- Verify security permissions on multiple users
- Fill out web forms
- Verify data in various forms and formats
- Acceptance and Regression testing of our web app with full logging
- Performance testing

What do **YOU** need to do?

4. "Changes in Latitude, Changes in Attitude"

Sometimes reframing the problem statement can help you get to the final destination.

Creative brainstorming, problem-solving skills, and experimentation can all help achieve the wanted outcome.

- Price
  - Examine the free options.
    - Shell scripting
    - Scripting languages with libraries
      - Perl
      - Python
      - Ruby is my language
      - Watir - used heavily for web testing.
- Support
  - Gain trust
    - Don't try to solve all the problems in one swoop
    - Show a series of successes.
    - Leverage those small victories to larger ones
  - Show the ROI over time.
    - Keep track of how long it takes to do something manually.
    - Keep track of how long your coding activities take.
- Experience
  - Invest in what you know best - yourself.
    - Learn what you can from others
    - Google is your friend.
    - Check out the unit tests shipped with the Ruby libraries (gems)
  - Build on small victories
    - Build up your toolbox.
    - Try to recognize patterns.
  - Don't be afraid to try something new

Example of a creative solution to a task not easily “automated”

- Operating System and software installs problem
  - I used virtual machines running under Windows Server 2008 to help.
    - Manual setup once per OS/software install combination
    - Now a lot easier to spin up that combination in the future
    - Saves time and effort down the road

5. “It’s My Job”

In my opinion, one of the strengths of using Ruby is in that it is an interpreted language. You can use the Interactive Ruby shell (irb) and start investigating the language and its libraries (called gems in Ruby-ese). The following Ruby example can be cut and pasted into irb and used as the basis of your experimentation. You will need to install the following gems

- Watir
- dbi (Database Independent Interface)
- dbd-odbc (Database Driver – Open Database Connectivity)
- Faster-csv

Pointers to more information for the above gems are included in the references. There is also a link to a Watir tutorial contained within the references. The Watir example page referenced in the below code is a

good place to start to familiarize yourself with some of the things that Watir can do, and because Watir is built upon Ruby, you have the entire Ruby interface available to use. The possibilities are endless.

To use the SQL example, you will need to use a custom connection string to match your database and login credentials. You may want to modify the actual SQL query to something that would return results from your database as well.

To use the Faster-csv example, you should have it point to a CSV file that contains a header row with at least two columns – 'TestName' and 'URL'. Or feel free to modify the code to use the headers in your own CSV file.

You may wish to enter commands into irb one line at a time. This way you can experiment with syntax and see the results of your handiwork right after you hit the <ENTER> key!

```
#  The hash mark at the beginning of a line marks the line as a comment.
# This script uses the following libraries – called gems in Ruby-ese
require 'Watir'
require 'dbi'
require 'Faster_CSV'

#---- Use Watir to fill in a web form
# Using the IE developer tool bar or Firebug to investigate your web control is very useful here
# The web page pointed to here is actually part of the Watir tutorial!
browser = Watir::Browser.new
browser.goto("http://bit.ly/watir-example")
browser.text_field(:name => "entry.0.single").set "Watir"

#---- Use Ruby to interact with SQL Databases
sConnect = 'DBI:ODBC:Driver={SQL
Server};Server=MyServer;Database=MyDB;Trusted_Connection=yes;'
hDBI = DBI.connect(sConnect)
sSQL=hDBI.prepare("select companyname from company")
sSQL.execute()
sSQL.fetch do |row|
      puts 'CompanyName  --  '+ row[0]
end
sSQL.finish()

#---- Use FasterCSV to interact with CSV files
# Use a CSV that has a header row
aList=FasterCSV.read('myFile.csv',:headers=>true)

# Iterate thru each line in the file and print out the test case name and target URL
aList.each {|aPage|
  sName=aPage['TestName']
  sURL=aPage['URL']
  puts sName + " – " + sURL
}
```

Pointers to the documentation for each of these libraries are contained in the References section. Also of use are the unittests that come bundled with each gem. The directory where you gem is installed should contain a directory named unittests. On my machine, I can find the Watir unittests at C:\Ruby\lib\ruby\gems\1.8\gems\watir-1.8.1\unittests. The unittests are all written in Ruby. Coupling the unittests with irb has given me great insight into how to better use the Watir gem.

6. "Stories We Could Tell"

Each company I have worked for has been a unique situation, each with its own set of challenges. Yet, the common thread with each is to figure out how to solve a problem with a suitable solution. Start off with a goal, and work your way towards it.

At Thomson Financial, test automation was not frowned upon, but it was not on the forefront of people's minds. Testing was thought of as a manual process. I found a project that could benefit from test automation and worked out a proposal for my bosses to why the project would benefit from automated testing for this data migration project. While the solution was written in perl, it could have been implemented in a number of different ways. I was familiar with perl and was confident that a solution could be crafted in a reasonable amount of time.

Thomson Financial

- Not originally slated for automated testing.
- Goal: Verify data conversion of historical stock quotes to the Euro.
- Not a web app, but a data migration that was under test.
- Original plan was to have someone randomly sample from the 40 million values
  - Figure out to convert one price point and compare it to the "official" number.
    - All the price points for a single stock.
    - All the price points for stocks that used that currency.
    - All the price points for all stocks against all currencies.
- Asked for month to go ahead and develop this from my QA Manager
  - Hesitant at first when I said a few weeks of effort needed
  - More receptive when we examined
    - Time taken to validate one value of one stock price
    - How easy it was to make a mistake in manual validation
    - Risk of inaccuracy encountered after the conversion went live.
- Wrote a framework in perl that examined **ALL** data points not just some of them
  - Found errors in the original conversion formula used
  - Found rounding errors in 2% of the conversion
- Entire effort took about 3 weeks
  - Design and implementation of the test harness
  - Test and retest phase
  - Included all logging and messaging.
- Key wins:
  - Earned trust at Thomson Financial
  - Presented results at Quality Week 1999

At Switchboard, some automation was already in place, but there was a project that accounted for 1/3 of our overall revenue. Making sure that this project was as bulletproof as possible was job number one for me. As such, I dove deep into my bag of tricks to craft a solution that would be reliable to preserve this income stream. Here, the automated solution also included the creation of a new web page that another test tool could drive. The web page I created was a simple web form that interacted with a known API. By using this web page, other tools could rapidly test the API's used by our customers by using web calls rather than low level C++ calls. I was able to leverage this web page to help craft the calls that would be used in load testing. In essence, a single page that I had created using HTML was an integral piece of the functional and performance testing portions of my job.

Switchboard

- Money not an issue, the challenge was testing against a known API that we defined.
- Company was willing to spend money to replicate the production environment for use of development and QA
- We had an expired license of SilkPerformer to update
- API could be construed as query string calls to a web server.

Goal: prove the functionality and reliability of an in-memory database used for searches

Many combinations of query string parameters to examine
I created a test page in HTML for manual queries for basic testing
Each query string parameter was represented on this HTML page
This formed the basis of a list of URLs that could be thrown against the web server manually
Used by both dev and QA.
Perl was used to create many more URLs for testing
Creation of the URLs
Persisting them to a file or files
SilkPerformer was used to drive the testing
The engine that read the files and threw them against the server
Used their built in logging and reporting tools.
Used for load, performance, stress, and failover testing.

Searches completed on the order of microseconds on the in memory databases.
Searches were returned on the order of subseconds on the webserver
Used by a customer that provided Switchboard with over 33% of all revenues.

Key wins:
Gained more trust from my co-workers
The customer never complained about the service offered by Switchboard.

At Compli, the challenge was different. There were no formal QA processes in place. There was no documentation about how the web application should behave. No money was available to buy tools. The C level officers were a little wary of automated testing. As the only QA engineer at the company, I had a challenge in front of me. How did I bring the use of automation into this company? It was at Compli that I dropped perl in favor of using Ruby. Ruby became my go to language because of the Watir gem. Watir comes bundled with a suite of unit tests that I used as examples to experiment with. It is a reliable tool that has become the cornerstone in al of my web testing efforts. Best of all, this tool fit the budget that I was working with.

Compli

Company had no real QA processes in place - ever.
Previous releases were painful on the company.
Buggy code released to production
Many phone calls with customers upon rollout
Many hotfixes created to address shortcomings
We had no money allocated for any testing tools.
Management didn't have a good idea of what could be accomplished.

Goals:
Make releases less painful for the company
Be able to turn around builds quickly
Define and implement a set of automated acceptance tests
Show the value of automated tests

I asked various people what was currently being verified with each release
- Customer support
- Internal channel support
- Basically one person from every department
Creation of a manual acceptance test plan
Went back to the people queried for input and comments.
Gained the trust of these people in the process
Used Ruby to implement the test plan

Initial implementation was only 40 scenarios with 200 verification points
Used Watir to drive the browser
Used ODBC to execute SQL queries as needed
Used Test::Unit as the test framework
Used the log4r library for logging purposes
Constructed a reusable automated testing framework that could be expanded

Key wins:

After the first rollout, both the president and CEO dropped by to say that this was the smoothest release ever in the history of Compli.
I gained more of their trust in the process of several releases.
I learned how to use Ruby.

Now we have an acceptance test framework that examines over 1200 test points
Takes about 2 person days to execute the test cases manually
Takes about an hour to run in the automated framework
Base regression test suite that verifies over 11000 test points
Takes about 3 hours to run to completion
More detailed regression tests also available
Multi day tests touching more edge cases
Run as needed.

More recently was a project that I was involved in this past May. We had a new integration with a 3rd party web site where we created reports based on the data that they provided. It entailed matching data against existing users in our system. The challenge was that sometimes the data didn't map to the users 100% of the time and we needed to be able to examine those unmatched data points to examine why we could not match the data. Was the problem in our matching algorithm or simply that the data really could not be matched.

The following project is my favorite example that illustrates the use of test automation in a non-traditional sense. The solution I crafted is not a 100% push button solution but illustrates how I used different tools to make the testing effort more automated than the straight forward (but error prone) manual verification. The solution implemented also did the verifications in a way that it would easy to verify new builds down the road.

Compli, week of May 16th, 2011

Goal: Verify our reports against the data generated by a 3rd party site (system of record).
Ideally: our reports would directly match the 3rd party reports.
Data Import
Users records to match against
3rd party incident reports from external site
Imagine our surprise when our reports didn't exactly match the 3rd party reports!
Reconcile report of unmatched data – one of two outcomes
Missing user records
Bad/missing drivers license data
Process every line on the exception report against our system
Reconcile our report vs. their report.

Test attempt 1

800 unmatched user records were found
Timed 3 records manually to estimate how long verification might take.
Entailed going thru the UI to verify the problem.
30-60 seconds per row to hand verify
Estimate of 38400 seconds (10.6 hours) to verify all 800 records

"Automated" Solution
- Export the unmatched data report from our app to Excel - manual
- Convert the Excel report to CSV – manual
  - Includes running an Excel macro to handle double quoting all fields
    - Excel save to CSV doesn't do this automatically for all fields
    - Poses problems when "numbers" are not interpreted as strings
- Formulate SQL queries to run – scripted in Ruby and ODBC
- Compare **every** line on the export file to a SQL query – scripted in Ruby

- Spent about 6 hours writing and testing the above process to help the verifications
  - Converting Excel to CSV issue
    - Data with leading zeros got munged and interpreted as integers
  - Add in
    - Debug lines
    - Output Logging
    - Better Comments
  - Made it more generic to allow greater reuse
    - Allow company name to be passed in
    - Made the SQL handle this passed in parameter

- Script took less than a minute to run

Test attempt 2
- Bug fixes came in to QA.
- 250 records to verify after bug fixes
- Guess how long it took to verify this record set.
- Now we were able to show the customer how many data issues were present in the system of record
  - Missing or extra leading zeros in ID fields
  - Missing or extra spaces and dashes on ID fields
  - Spelling input errors on text fields.

7. "Don't Bug Me"

Watch out for cases where you get so wrapped up on things that you can lose sight of what you're actually trying to accomplish. Watch out for the following traps

Don't lose sight of your day job
- If your day job is to create automated tests, you'll be in better shape.
- If your job is to manually test, don't forget what you're getting paid for

Don't lose sight of the original goal
- Focus on what needs to be done.
- Don't get sidetracked by things that are "cool"

Don't take comments personally.
- Now you're in the position of developing software.
- Others will give valuable input.
- Listen to them

Don't build something you don't really need just because it's cool.
- Create tools that are useful
- Create tools that accomplish the stated goal

KISS

Don’t get stuck on a problem
- If you do get stuck, you need to get unstuck
- Maybe someone else has a different perspective on things
- Comments from others can give you a different perspective to the issue at hand
- Maybe a problem reframe is needed

Don't bite off more than you can chew
- Recognize potential problem areas
- Ask for help when its really needed

Don't just cut and paste code
- Understand what the code is doing and why.
- If you don’t understand the code, you should not be using it.
  - How will you maintain it when problems arise?

Don't try to automate something that should not be automated.

Candidates for the “Do not automate” pile
- Cost/time savings will not appear
- When the script maintenance cost will be too high
- A really complex one-off test scenario
- Unstable user interface in use
- Testing tools unavailable or difficult to use

8. "Cheeseburger in Paradise"

Hopefully these ideas will serve you well in breaking down the barriers to getting automation into your company. Trust, communication, and documentation are some of the keys to getting what you need to accomplish this.

Figure out what the goal is
- If you can’t define what you want, how do you know what to do?

Examine how you can break up the problem space into discrete pieces that you can solve
- What do you already know how to do?
- What will you need help with?
- What new things will you have to learn?

Use whatever will solve the problem.
- Using the same language will be easier, but it's not the end of the world
- As you become more successful, settle on the technology side of things

Recognize patterns when you can.
- If you are seeing something being done over and over again, try to generalize the steps for reuse.

Be sure to comment the scripts!
- It helps with maintenance.
- I used to be a perl guy. Now my tricks have been reimplemented in Ruby
- By commenting the perl scripts, I knew what I needed to accomplish in Ruby
- Makes it easier to hand off to someone else for use
- Commented scripts can be used for training purposes.

Show the ROI
- Time taken to create the scripts vs. the time to run them manually
- Show your managers that this is time well spent

Build on your success
- Success will beget more success
- Build up the trust factor and things will get easier down the road

References

Ruby - http://www.ruby-lang.org/en/

Ruby Quickstart - http://www.ruby-lang.org/en/documentation/quickstart/

Watir - http://watir.com/

FasterCSV - http://fastercsv.rubyforge.org/

DBI - http://www.tutorialspoint.com/ruby/ruby_database_access.htm

ODBC - http://www.ch-werner.de/rubyodbc/

Microsoft Powershell Scripting - http://technet.microsoft.com/en-us/scriptcenter/dd742419

Microsoft Command line reference - http://technet.microsoft.com/en-us/library/bb490890.aspx

OpenSta - http://opensta.org/

Selenium - http://seleniumhq.org/

Jimmy Buffett - http://www.margaritaville.com/

Unit Testing a C++ Database Application with Mock Objects

Ray Lischner
rlischner@proteuseng.com

Abstract

Sometimes, an idea isn't valuable because it's new but because it's old. Unit testing isn't new. Databases aren't new. Database client libraries aren't new. Writing a database client library that directly supports unit testing via mock database classes and the injection of mock results—that's valuable.

Unit testing is not new, and has long been a common element of software testing. Various design and development techniques, such as extreme programming and test-driven design, have reinvigorated unit testing as a best practice. In order to unit-test an application effectively, the developer must be able to isolate the code under test. The most common method of isolating code for unit testing is to use mock classes or objects to stand in for other classes, especially external services and interfaces.

Java and similar languages offer language support for easy definition and construction of mock objects. C++ presents additional challenges. Some C++ utilities help you write mock classes, but in order to unit-test effectively, a C++ library must be designed for testability. Given the evident value of unit testing, it is unfortunate that so many important libraries are not designed with unit testing in mind.

This paper presents a new C++ database client library, one that has been designed from the start with unit testing as a primary design goal. This library uses abstract interfaces to enable the developer to follow a clean database interface, and to substitute a mock implementation for unit testing and a real implementation for production use. The application code needs to be compiled only once, so you can be sure that you are unit testing the real application code, even when linked with the mock database library.

The resulting library enabled delivery of a database application with zero known defects. The paper presents the design decisions and shows how testability drove the library design, and how the library and resulting applications benefited from the emphasis on unit testing.

Biography

*Mr. Lischner has been designing and developing software for about three decades, starting at the California Institute of Technology, where he earned his B.S. in computer science, and subsequently at large and small companies on both coasts of the United States. Meanwhile, he earned his M.S., also in computer science, from Oregon State University and performed stints as an author (*C++ in a Nutshell*,* Exploring C++*,* Shakespeare for Dummies*, and other books), consultant, teacher, and stay-at-home dad. Many years ago, Ray served on the PNSQC board and various committees. He currently develops software for Proteus Technologies, where he holds the title of Distinguished Member of Technical Staff. Ray is a senior member of IEEE and a member of ACM.*

1. Introduction

This paper introduces a database client library, unimaginatively named *libdb*, that was deliberately designed and written to support unit testing of database applications. None of the ideas introduced in this paper is new, novel, or different. The combination, however, is unique. We hope that this case study can serve as a model for other C++ service libraries to support unit testing of applications.

Before examining the library in depth, a review of unit testing in general is in order.

1.1. Unit Testing

Unit testing, or testing in the small (Hetzel 1988), is testing of the smallest pieces of testable code (the unit). Typically, a unit is a single member function or free function because in C++ it isn't possible to execute a single statement outside of any function. Unit testing is a form of developer-led testing (McConnell 2004), and is just one of many layers of testing that a typical development team performs. The immediate benefactor of a test failure is the developer, who can quickly fix the code (or the test).

A key distinguishing feature of unit testing is isolating the Code Under Test (CUT). When testing a single class or a single member function of a class, you want to ensure that you are testing only that class or function, and not other classes or functions that the CUT relies on. Therefore, you want to design your code to reduce coupling between classes, which is a desirable trait independent of unit testing (Weinberg 2008, Briand 1997), and you want to be able substitute simpler classes to emulate the classes—called *mock* classes—that surround and support the CUT (Mackinnon 2001).

Further isolating the CUT is to remove all ties to the outside world, in particular, ties to external databases. The ideal unit test has no connections to outside services, no files to read or write, no user interaction, and no ambiguity. A test passes or fails, with no other states in between. One advantage of unit tests over other kinds of testing is that a unit test is often the best way to check error conditions. A unit test can set up artificial error states and verify that the CUT performs as designed.

Unit testing has been around for decades (Hetzel 1988), and long been considered one of the key best practices of software development. With the advent of extreme programming (XP) (Beck 2004) and test-driven development (TDD) (Beck 2002), unit testing gained renewed prominence. The value of thorough unit testing is now clear and well-supported by evidence. Furthermore, TDD has been shown to be effective as a technique to facilitate the delivery of high-quality software by ensuring the software is testable and well-tested.

1.2. Unit-Testing Frameworks

It is no surprise, therefore, to learn that a variety of libraries provide unit testing frameworks. In C++, we have CppUnit (2011), Boost.Test (2011), Google Test (2011), and many others. These frameworks share a number of common attributes. You can:

- define a suite of independent tests;
- write a test as a series of assertions about the state of the code and test environment;
- craft a test-runner program that can invoke one, some, or all of the tests; and
- set-up and tear-down the test environment uniformly for all tests.

Each framework has its individual advantages and disadvantages; the details are not germane to this paper. The examples in this paper use Boost.Test, but any framework is acceptable.

1.3. Mock Classes and Objects

A mock object (Mackinnon 2001) is an object that supports unit testing by replacing the CUT's neighbors with simple objects that record their interactions with the CUT and by providing mock results to the CUT. In this way, the CUT is isolated from other classes and functions, so you can be sure that your unit tests are truly testing your unit, and not some neighboring, or even distant, unrelated code.

For example, suppose the CUT opens a file, reads some text from it, and closes the file:

```
std::string readline(std::string const& path)
{
   try {
      File f(path);
      std::string result( f.read_line() );
      f.close();
      return result;
   } catch (PermissionError const& ex) {
      return "No permission";
   } catch (FileError const& ex) {
      return path + ": " + ex.what();
   }
}
```

How do you test such a function? Maybe you create a bunch of files, one empty, one with one line of text, another with multiple lines of text. What about error conditions? What happens if the file permissions do not permit reading? You can test these conditions by creating the right kinds of files. But what about other conditions?

What if an I/O error interrupts the read mid-line? Will your code catch the right exception and handle it the right way? That's where mock objects come in handy. Instead of using the real `File` class, use a mock implementation of `File`. The `MockFile` class lets you specify the exact behavior of the object, from exceptions that it throws to results that each function returns. So you can write tests such as the following, which arranges for the `File` class to throw an exception, and then tests that the `readline()` function catches and handles the exception correctly.

```
void test_readline_errror()
{
   MockFile.set_exception(FileError("mock error"));
   BOOST_CHECK_EQUAL(std::string("filename: mock error"),
                     readline("filename"));
}
```

Using the `MockFile` class instead of real files eliminates a dependency on external files. You don't run into the situation in which someone accidentally renames a file, and suddenly your tests stop working.

In languages such as Java, mock objects are commonly created by frameworks such as Easy Mock (Easy Mock 2011), jMock (jMock 2011), and others. The framework allows you to create mock objects according to your tests' requirements, rather than hand-crafting mock classes to emulate your classes.

Using mock objects in C++ is harder due to the richer compilation environment. It isn't enough to create mock objects on the fly, based on reflection. C++ also has templates, requiring additional complexity from any mock framework. On the other hand, some of the common C++ mock frameworks impose restrictions, such as the use of pure abstract interfaces, such as Automatic Mock Object for C++ (amop 2011) and Mock Objects for C++ (mockpp 2011). For these reasons, the use of mock objects and classes is less common in C++ than in Java.

Most C++ programmers cope by ignoring the issue. Well-designed code exhibits loose coupling between classes, so pure isolation of the code under test is not critical for successful unit testing. But when the code relies on external services, such as database access, the lack of mocks or other isolation techniques presents a serious challenge. With the dearth of general mock frameworks, it is incumbent upon the authors of third-party libraries to ensure their libraries support effective unit testing.

Unfortunately, such libraries are rare. This leaves the programmer in a quandary. You know that unit testing is important. You know that designing your application for testability will result in a higher quality design. You know that writing unit-tests while you write the code is an effective, efficient means of designing and developing software.

So when you need to use third-party C++ libraries, such as database client libraries, how do you proceed? The next section takes a look at how the author coped with this situation.

2. Database Libraries

The author recently needed a database library for writing a small database application. Such libraries are common—for example, OTL (2011), SOCI (2011), and SQLAPI++ (2011)—but not one of these libraries offers direct support for unit testing. Many developers face such restrictions by foregoing unit testing, relying instead on big-bang testing of the completed database application.

Forward-thinking developers may try to apply a modified form of TDD by incrementally developing the application, but this approach is still limited because the application must always interact with a real database. When the real application interacts with a real database, for example, how easy is it to test that the application rolls back correctly when the network connection is disrupted in the middle of the transaction? Even a library that supports an in-memory database is not designed to allow this kind of testing.

Another drawback to live-database testing is to ensure that multiple developers can run tests simultaneously. Thus, each developer needs a distinct database instance, and indeed probably needs multiple database instances. Another project that uses a full MySQL installation to perform “unit” tests takes a full sixteen minutes to run through a few dozen tests. This is after heavy optimization reduced the time from twenty-five minutes. Ideally, unit tests should run much faster than twenty-five or even sixteen minutes. A number of development teams require unit tests to pass prior to committing changes to the source code repository. In such an environment, unit tests that are burdened by real database connections would be infeasible.

Faced with the abundance of evidence that real unit-tests are necessary for effective development, the author sought an existing database library that lent itself to effective unit testing, but found none.

3. Mocking a Library in C++

When faced with the need to substitute a mock library for a real one, the author first tried to write a mock library with the same API. By directing the compiler to the mock headers instead of the real headers, the CUT was compiled for unit testing. When linked with the test-runner, a working unit-test was created. This approach has several drawbacks:

1. The CUT must be recompiled with the mock headers instead of the real headers. This doubles the compilation time, which may be a factor on large projects.
2. The CUT is not the real application code. By compiling with different headers, the mock library can introduce or mask errors. For example, we successfully hid a memory leak in our code by deleting an object in the mock library that was not deleted in the real library. (The mock library implemented the API incorrectly, but the error sneaked past our code review.)

3. The library may not lend itself to unit testing. Usually, this means the library is probably hard to code to, but the application developer may not have a choice of library. If the library had been designed with unit testing in mind, it would probably have had a better API.
4. Unit testing and production code must never be linked together. That is, suppose the application has two object files. If file A is compiled for unit testing and file B is compiled for production, if you mistakenly link files A and B, the results would be disastrous. In C++ terms, this would violate the one-definition rule (C++ 2003, [basic.def.odr]), resulting in undefined behavior. In practical terms, the result was usually a segmentation fault. Careful attention to build environments can prevent this kind of error, but in the author's experience, carefully designed and executed build environments are uncommon.

With a simple library, it may be possible to craft a mock library that has the same headers as the real library, but with different object files. In this case, recompilation is not necessary—only relinking. With a modern C++ library, however, this is rare. It is highly unlikely that a useful C++ library would entirely eschew templates and inline functions. It is even less likely that an accomplished C++ developer would want to use such a primitive library.

When faced with the database client library dilemma, the author decided that the best solution would be to write a new database library from scratch, this time, designing the library explicitly to support unit testing. The goals were as follows:

1. The library must facilitate unit testing. The application developer must be able to emulate a rich variety of error conditions that are hard to reproduce in an integration or system-level test.
2. The real application code must be unit-testable, without the need for recompilation. That is, when the database application is compiled, we should be able to use the same object files in a unit test that we do in production. We may choose to compile with more debugging and less optimization during testing, but the option is always available. In unusual, but not-rare-enough cases, we encounter compile defects that are difficult to track down if they never manifest during unit tests.
3. The library must be easy to write. The customer's requirements were for the database application, not a database application plus a reusable library. I had to be sure that writing the application and the library would together cost less than writing just the application (taking into account the extra time that would have been required due to the lack of unit testing).

I decided to use abstract interfaces for the library. The abstract interface would define the application programming interface (API). A mock implementation would be used for unit testing. Another concrete implementation would be used for the real database. We used only MySQL at the time, but other concrete implementations could support other databases.

4. Abstract Interfaces in C++

C++ can do anything Java can do, but sometimes it's harder in C++ than in Java. As we've already seen, implementing an on-the-fly mock object framework is harder to accomplish in C++ than in Java. But abstract interfaces are just as easy to write in C++ as in Java. C++ also has the advantage that interfaces do not have to be pure. Sometimes, it is advantageous to write a class that is mostly an interface, with just a hint of implementation thrown in. The database library has a couple of instances where impure interfaces provide clear benefits.

In C++ an abstract interface is a class with pure virtual functions, written as follows:

```
class this_is_abstract {
  virtual ~this_is_abstract();
  virtual void function() = 0;
};
```

The = 0 tokens tell the compiler that the function has no implementation, and that a derived class must override the function. The compiler prevents any code from constructing an instance of a class that has a pure virtual function. Instead, the developer must derive a concrete class that overrides and implements every pure virtual function. The compiler allows construction only of concrete classes.
The database library is defined primarily in terms of abstract interfaces. In particular, the `db::sql` and `db::statement` classes are abstract interfaces to represent the main interface to a SQL database and to a prepared statement, respectively. The database application code (almost) never refers to any concrete classes, using only the abstract classes.

The only time the application code names a concrete class is when it instantiates a class that derives from `db::sql` to kick things off. In our application, we have a small `main()` function that instantiates the necessary application objects as well as the `db::mysql::sql` object.

We never unit-test `main()`. Instead, we put all the application logic in application classes, and unit-test the application classes. Most of our applications have abstracted `main()` to the degree that we encapsulate it in a single macro that just names the main application class, and any concrete classes that we wish to inject into the application code. In this way, we have nothing to test in `main()`.

Unit tests create a `db::mock::sql` object and hand it off to the application code under test.

5. Design of libdb

Although we used only MySQL, I tried to keep the design flexible to allow for other concrete classes. I implemented the mock and the MySQL concrete implementations in parallel. Sometimes, the mock requirements drove design decisions, and other times MySQL drove design decisions. In both cases, I ensured that the decisions were consistent with the other implementation.

The main interface is the `sql` class. It represents a single database connection. An application can have any number of connections in any number of threads. The `sql` class manages transaction commitment and rollback, and it creates prepared statements, which are represented by the `statement` class. Figure 1 illustrates the class hierarchy.

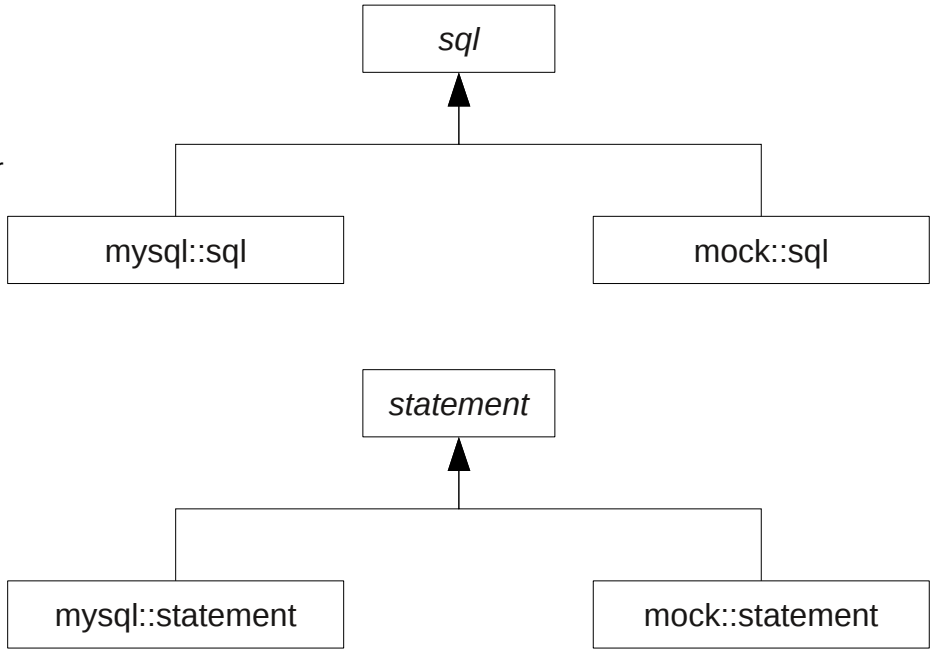


Figure 1: Class hierarchy

A `statement` object lets you bind rvalues to parameters, bind result lvalues, execute the statement, and fetch result records directly into the bound variables. (For those not conversant in C++-ese, think of an lvalue as a variable, and an rvalue as an expression. For those who are C++ experts, please forgive my oversimplification of these terms.) I made the decision early that the goal is to support specific applications, not general-purpose tools. Thus, there is no interface that returns a row as

an array of values to be queried by name or index. Instead, you must know exactly what the query will produce and bind lvalues of the correct type.

For example, suppose the *person* table contains `first` and `last` columns for the person's name, and `id` column for a primary key. The code to look up a single person's name might look like the following:

```
std::string const get_name_query(
      "SELECT first, last FROM person WHERE id = ?");

class person_app
{
public:
  person_app(db::sql& sql) :
  sql_(sql),
  get_name_(sql.prepare(get_name_query))
  {}

  std::string get_name(int id)
  {
    std::string first, last;
    get_name_.bind_param(id);
    get_name_.bind_result(first, last);
    get_name_.execute();
    if (get_name_.fetch())
      return first + ' ' + last;
    else
      throw not_found("No such person: ", id);
  }
private:
  db::sql& sql_;
  std::scoped_ptr<db::statement> get_name_;
};
```

The `bind_param()` and `bind_result()` functions are template functions that are implemented in `db::sql`. The application calls `bind_param()` once, if necessary, to bind all parameters (corresponding to the question marks, ?, in the text of the prepared statement). Each function argument corresponds to a ? in the query. The library performs basic error-checking to ensure the correct numbers of values are bound.

In the db::statement class, there are overloaded `bind_param()` functions, with a different number of arguments. They all follow the same form, such as:

```
template<class T1, class T2>
void bind_param(T1 const& arg1, T2 const& arg2)
{
  params p;
  p.push_back(param(arg1));
  p.push_back(param(arg2));
  bind_params(p);
}
```

The `bind_params()` function is virtual. The MySQL implementation constructs a `MYSQL_BIND` instance for each parameter and maps the C++ type to the equivalent MySQL type. The type `std::string` is mapped to a MySQL string, for example, and `std::vector<unsigned char>` is the libdb storage for BLOBs (Binary Large OBjects).

If the statement returns a result set, the application must call `bind_result()` to bind a variable to each column of the result set. The library automatically assigns values to these variables when the application calls `fetch()`. Binding of results is slightly more complicated than binding of parameters because strings and BLOBs have varying length. Thus, libdb must determine the result size, then resize the lvalue that was bound in the call to `bind_result()`, and only then it is safe to copy the result to the bound lvalue. Thus, the caller simply binds, say, a local variable of type `std::string` to a column of SQL type VARCHAR(255). The `bind_result()` function tells libdb about the variable. When libdb fetches a result row, it calls the variable's `resize()` member function to set the string to the correct size; then libdb copies the result string into the variable.

The `bind_result()` function looks very similar to the `bind_param()` function, except it constructs a `db::result` object for each argument instead of a `db::param` object:

```
template<class T1, class T2>
void bind_result(T1& arg1, T2& arg2)
{
  results r;
  r.push_back(result(arg1));
  r.push_back(result(arg2));
  bind_results(r);
}
```

The template functions determine the argument types and values, and prepare the parameter or result in a general way before passing a vector of `param` or `result` objects to `bind_params()` or `bind_results()`, both of which are private, abstract functions that perform the actual binding. In this way, the impurity of the `db::sql` class avoids duplication of common code in each concrete class. Using template functions in the abstract class improves clarity and results in higher quality of the database library.

In the mock library, a unit test prepares the mock results by storing responses. A response is a list of parameter values that libdb copies into result variables to mimic a database query. Because an application can make several queries inside a single function, the unit test must be able to store multiple rows in the mock result set. Furthermore, the test must be able to manifest various error conditions. The key is the `store_response()` function.

An example of a unit test for `get_name()` is as follows:

```
void test_get_name()
{
   db::mock::sql sql;
   person_app app(sql);

   sql.store_response(get_name_query, "Jane", "Doe");
   sql.store_response(get_name_query, db::mock::end_of_result_set);
   sql.store_response(get_name_query, db::mock::error);

   BOOST_CHECK_EQUAL(std::string("Jane Doe"), app.get_name(1));
   BOOST_CHECK_THROW(app.get_name(2), not_found);
   BOOST_CHECK_THROW(app.get_name(3), db::exception);
}
```

The `db::mock::sql` class implements `store_response()`, which is how a unit-test prepares the mock result sets for the application under test. Result sets are keyed by the text of the prepared statements. Each statement text has associated result records that the mock `fetch()` function fetches and binds to the result lvalues in the application code.

Thus, the mock `sql` object stores the strings "Jane" and "Doe". When the test runs, the mock `statement` retrieves the values and copies them to the first and last variables in the application code. As far as the application is concerned, it executed a query against a database and retrieved the first record of a result set. It doesn't know or care that the result set is just an array of values in the mock `sql` object.

The `store_response()` function can also store special values to represent the end of a result set or an error condition. When the mock statement object fetches one of these special values, it causes `fetch()` to return false (indicating the end of the result set) or to throw an exception. In pseudo-code, the mock fetch() function looks something like the following:

```
bool mock::statement::fetch()
{
  if (no more responses)
    return false;
  else if (response is pointer and pointer == mock::end_of_result_set)
  {
    ++response_iterator_;
    return false;
  }
  else if (response is pointer and pointer == mock::error)
  {
    ++response_iterator_;
    throw db::exception("mock database error");
  }
  else
  {
    copy_response(*response_iterator_, results_);
    ++response_iterator_;
    return true;
  }
}
```

Note how the mock statement keeps an iterator into the sequence of responses. Each time the iterator advances, it moves from one call to `store_response()` to the next. If the iterator points to a response of one element of pointer type, it checks the pointer value to determine whether that pointer matches a magic indicator. Otherwise, it copies the response to the bound result variable.

Here, the requirements of the mock library drove a critical design decision. We decided to store mock responses using the `param` class because it already implemented everything we needed for storing typed data. The only question was how to represent the magic indicators, such as `db::mock::error`. We could not use distinguished values because we would not be able to distinguish them from legitimate query responses. We decided that the best way to represent an indicator was to use pointers. We define a private object, and set `db::mock::error` to point to that object. The pointer value is unique and unambiguous. The requirement the mock library imposed on the `param` class, therefore, was to present pointer values.

The initial implementation of `param` stored a copy of the data. This seemed logical because it freed the developer from any concerns about lifetime of the bound value. In order to handle magic indicators, however, we had to change `param` to store pointers. This has the additional benefit of avoiding copying of strings and BLOBs, resulting in a performance gain, too.

In practice, lifetime issues are nonexistent. Our usage scenarios always place the binding with `bind_param()` in the same function as the calls to `execute()` and `fetch()`. Thus, it turns out lifetime issues were not an issue.

I wrote three test cases in one function for the sake of brevity. An alternative approach is to write three distinct unit tests for the three separate cases: found, not found, error. The advantage of having each test case in its own function is that most unit-test frameworks give you the ability to control which test functions to call, but not the ability to choose individual test cases within a function. Also, some unit-test frameworks (such as CppUnit) fail the test function if any assertion in the function fails. Test cases in the same function that follow the failure never run. Thus, you might not learn about a failure until after a complete editing cycle to fix another problem.

Sometimes, however, a unit test requires a lot of code to set-up the test. One setup may suffice for a series of related test cases. In that situation, it is often easiest to write a single function with multiple test cases. Whenever possible, however, do your best to isolate test cases in separate test functions.

6. Maintenance

Of course, software never stands still. Any change to the API requires updating the interface (such as `db::sql`) and the concrete implementation classes (`db::mysql::sql` and `db::mock::sql`). Fortunately, the costs are not triple because writing the abstract interface and mock implementations are usually much easier than writing the real, vendor-specific implementation. Once you know how the vendor-specific class works, it is usually easy to write the corresponding mock class and abstract interface.

The compiler even helps you by catching most omissions and mistakes, such as forgetting to implement a new member function or changing a function signature in a concrete class but not the interface.

It is feasible to write a tool that could extract the abstract interface for you, but we do not have such a tool. If we created many abstract interfaces, it would be worth acquiring or writing such a tool, but we don't do it often. So far, the burden of maintaining the abstract interface has been small and manageable.

Another concern for overhead is the increase in size of the code base. Compiling C++ is never fast; compiling three files instead of one is nearly three times slower. On the other hand, an earlier generation of the database application required instantiating a database server in order to run even the simplest test. By replacing many tests with true unit tests that use a mock database, the overall time required to compile and test the software was reduced by an order of magnitude (20 minutes to 2 minutes).

7. Legacy Code

An unsolved problem is tackling legacy code. Sitting in our legacy code base, for example, is an application that communicates over a socket. This application is no different than thousands of other socket-based applications. But we don't have a mock-socket class in our toolbox.

All the existing tests for the legacy code must run real application servers, and open real sockets. When a test takes 1 second instead of 1 millisecond (which would be a slow test using mock objects), it really drags down the test suite. A typical application can have hundreds of unit tests. This particular application takes about ten minutes to run through all its tests.

This application is due for some major changes in the near future, but we keep pushing the project down the to-do list because we are all hesitant to tackle it. Ironically, the problem is that we already have a socket library. If we didn't, we would probably design one from scratch using the principles outlined in this paper. Because we have a socket library and many developers have experience using it, we are reluctant to introduce another socket library, even if it does help us with our unit testing. Somehow, we have to figure out how to introduce mock sockets into this legacy socket library.

If we figure out how to solve the legacy code problem, we'll be back with another paper.

8. Conclusion

Developers test code as a way to deliver higher quality results to our customers. The proof of any technique, such as using a mock database library, is whether it helps improve quality or reduce costs. In this case, we delivered our application on time with zero known post-delivery defects in the database portion of the application. During development and integration, the database portion was so reliable, we were able to use the MySQL database as our oracle for testing other parts of the system.

The database was a small part of the overall project, and we did not break it out as a separate schedule item. Thus, we cannot definitively say that development of the unit-testable database library reduced costs. But it certainly had no negative impact on the schedule.

With no negative schedule impact, and zero defects discovered in the product after shipping, we consider this approach to be a complete success.

We certainly do not suggest that abstract interfaces are suitable for all C++ libraries. Many uses of C++ are especially sensitive to performance, and the overhead of even one virtual function can be too much (or one virtual function table in a class that is instantiated millions of times). One of the advantages of C++ over many other languages is the richness of the language, allowing for flexibility and choice in a way that Perl programmers can appreciate. There’s more than one way to unit-test C++.

Other techniques for using mock libraries are possible. Each technique has its own drawbacks. We considered using compile-time substitution of a mock library, which complicates compilation and linking. We opted for abstract interfaces to avoid these issues and because we could live with the run-time overhead of a single virtual function call. For the database library, we are happy with our design decisions.

Adapting these techniques to existing libraries is much harder. If a library is not designed to be mockable, it can be hard or infeasible to introduce mock classes in support of unit testing. But other C++ service libraries can benefit from this approach, if you can find a way to introduce mock classes. Even if a different approach is taken, the idea of delivering a service library with a real and mock implementation can only benefit C++ application developers.

References

amop. 2011. *Automatic Mock Object for C++*. http://code.google.com/p/amop

Beck, K. 2002. *Test Driven Development: By Example*. Addison-Wesley.

Beck, K., and C. Andres. 2004. *Extreme Programming Explained, 2nd ed*. Addison-Wesley.

Boost.Test. 2011. *Boost test library*. http://www.boost.org/doc/libs/release/libs/test

Briand, L., P. Devanbu, and W. Melo. 1997. An investigation into coupling measures for C++. *ICSE '97 Proceedings of the 19th International Conference on Software Engineering*, pp. 412–21.

C++ standard. 2003. *ISO/IEC 14882:2003: Programming languages C++*.

CppUnit. 2011. *C++ port of jUnit*. http://sourceforge.net/projects/cppunit/

Easy Mock. 2011. *Easy Mock 3.0 Home*. http://easymock.org

Google Test. 2011. *Google C++ testing framework*. http://code.google.com/p/googletest/

Hetzel, B. 1998. *The Complete Guide to Software Testing, 2nd ed.* QED Information Sciences.

Jeng, B. and E. Weyuker. 1989. Some observations on partition testing. *TAV3 Proceedings of the ACM SIGSOFT '89 third symposium on Software testing, analysis, and verification,* pp. 38–47.

jMock. 2011. *An expression mock object library for Java*. http://www.jmock.org

Mackinnon, T., S. Freeman, and P. Craig. 2001. Endo-testing: Unit Testing with Mock Objects. *Extreme Programming Examined*. Addison-Wesley.

McConnell, S. 2004. *Code Complete, 2nd ed.* Microsoft Press.

Mockpp. 2011. *Mock Objects for C++*. http://mockpp.sourceforge.net

OTL. 2011. *Oracle, ODBC, and DB2-CLI Template Library*. http://otl.sourceforge.net/

SOCI. 2011. *The C++ Database Access Library*. http://soci.sourceforge.net/

SQLAPI++. 2011. *C++ library for accessing databases*. http://www.sqlapi.com/

Weinberg, G. 2008. *Perfect Software and Other Illusions About Testing*. Dorset House.

Playback Testing Using Log Files

Author
Vijay Upadya, vijayu@microsoft.com

Abstract

Most testers are familiar with record/playback testing. A variation of this approach is log playback testing. Instead of explicitly recording the user actions using a recorder, this technique simply relies on the information in the log file produced by the system under test to play back the scenario. This is extremely useful for reproducing issues reported by customers which are typically hard to reproduce otherwise due to lack of detailed repro steps. This approach also lends itself well to creating data driven tests and can be used to augment the existing test automation.

Testing and investigating failures in a non-deterministic software system can be very costly and painful. In most cases the only source of information to know what code path got exercised are the log files produced by the application. This paper talks about how the information in log files can be used to playback the sequence of actions to test specific code paths and also reproduce bugs. We will look at exploring an extension of this approach and how log files can be used as a source for data driven testing by simply varying the sequencing of actions that are played back. This paper illustrates the technique using real a world example. Finally the paper discusses the benefits and how it can be applied to different test domains.

Biography

Vijay Upadya is a senior software development engineer in test at Microsoft, currently working at the Microsoft main campus in Redmond, Washington. Over the past decade, he has been involved in software development and testing for products ranging from compilers to libraries like Linq To SQL, Silverlight and RIA Services to Windows Live mesh. Vijay primarily focuses on test strategy, test framework and tools and test frameworks for the team.

Vijay has a Master's degree In Systems Analysis and Computer Science from the Mangalore University, India

1. Introduction

Log files are typically used for diagnosing issues reported by users of the product. In most cases log analysis is done manually where developers read the log and try to understand the sequence of events happening during the failure and where it went wrong. For catastrophic failures like crashes the call stack will mostly suffice in figuring out the issue in the product. But when there are functional failures, having a detailed log will go a long way in helping diagnose the issue quickly.

Now we can take it further and design logging with the intent of generating logs for easy auto analysis and use it to actually playback the scenario as it happened in user's machine. In order for this to be successful, the logging format needs to be designed right from the beginning of the product cycle with playback in mind.

Log files are also a great source of information that shed light on how customers are using the product and information contained in them can be sourced as test cases that test teams can then run regularly during their test passes.

In the next sections we will discuss how log files can be used for playback and more exhaustive testing. We will use an example of file synchronization software[i] for the discussion.

2. Information needed for playback

In general log files for the application contain sequence of important events that happen along with information about data. Let us explore characteristics of a good log that is 'playback ready'.

There are two main pieces of information needed for playback

- User intent (i.e., scenario user intended to accomplish)
- Path taken by product in fulfilling this intent (i.e., product's execution flow)

2.1 User Intent

User intent is about capturing what a user intended to do while using the product. It could be a simple task or a series of tasks as part of a larger scenario.

Examples:

- User performing mathematical operations on a series of numbers in a calculator application
- User copying photos from camera to local folder on his PC and syncing it to other PCs using file synchronization software.

A log file needs to provide enough information to make it possible to interpret the user intent. There are two items of interest for capturing user intent

- Timeline of actions performed by user
- Data passed to each of these actions

For simple applications like a calculator, just capturing user intent will probably suffice as the component behaves in a deterministic manner for a given input. For example, user presses 2 + 4= and calculator produces 6. The calculation logic is synchronous and execution flow deterministic. However for non-deterministic software like file synchronization where there are a number of threads executing in an asynchronous manner, and where the order of operations are not guaranteed, just capturing user intent is not enough. We need an additional piece of data – the path taken by the product or execution flow.

2.2 Product Path

Path taken by the product: When a user performs a specific task, the product itself will perform a series of tasks to accomplish the user intent. These tasks are typically implemented in multiple components and some of these could be executed in parallel.

There are two items of interest for capturing product path
- Timeline of states and their transitions while executing tasks
- Information on any concurrent execution of tasks

3. How to capture user intent

In general, in order to capture user intent the product needs to understand general user actions or entry points to the application. This would be happening in any functional product. For example, when a user presses '+' in a calculator application, the application will 'interpret' it as an addition (user intent). In most cases it's just a matter of logging this information at appropriate locations in the product where it 'understands' the user intent. The timeline is also an important piece of data that enables determining the ordering of operations happening in the product and helps identify any concurrent operations happening. The product needs to log the timestamp information indicating when this action occurred along with any relevant data passed.

4. How to capture the product path

This involves thinking through control flow of the product while realizing the user intent and would involve logging different state changes and their transitions. Again here timestamp information needs to be logged along with data. More on this is discussed in the next section with an example.

5. How to playback

There are two things that need to be played back - one is the user intent and the other is the control flow of the application. For simple applications, just playing back the user intent (using the same data and in the same order) would help in replaying the scenario and reproducing the bugs. However for more complex products that are non-deterministic in nature (meaning, for the same scenario it could take a slightly different code path due to its asynchronous nature) the product path also needs to be 'played back' so that it converges on the path that caused failure in the user scenario.

Let's take an example of file synchronization software. There are three main operations that users of this application typically perform – Creation, Updates and Deletion (CUD) of files and folders. Now these can happen in any order at any time on any number of files on any number of devices that are enabled for sync. The product need to understand each of these main user gestures (CUD) and it's easy for it to log when it happens.

Looking into the control flow of typical file synchronization software while performing these CUD operations reveals that it comprises of the following sub-operations
- *Scanning* for detecting changes,
- *Uploading* (file to cloud storage),
- *Downloading* (file on another machine) and
- *Realizing* (copying) the file downloaded to right folder location so that the file appears on disk on other machine.

Some of these operations can happen in parallel asynchronously (for example, uploading and downloading).

Let's say a user reported a 'failure to sync' issue and shared the log file which looks something like this (only relevant portion shown for illustration purposes) with the log format 'TIME_STAMP' 'TASK' 'Data'

```
06-14-2011 00:08:46.507 SCANNER: thread for 'C:\Users\vijayu\SyncFolder' started
06-14-2011 00:08:46.617 SCANNER: FILE_ACTION_ADDED for 'C:\Users\vijayu\SkyDrive\ myPhoto1.jpg'
06-14-2011 00:08:48.703 SCANNER: thread finished scanning
..
06-14-2011 00:08:49.826 DOWNLOADIN file 'myPhoto2.jpg'
06-14-2011 00:08:49.827 UPLOADING file 'C:\Users\vijayu\SyncFolder\myPhoto1.jpg'
..
06-14-2011 00:08:51.626 DOWNLOADIN file 'myPhoto2.jpg'
06-14-2011 00:08:51.629 UPLOADING file 'C:\Users\vijayu\SyncFolder\myPhoto1.jpg'
..
06-14-2011 00:08:53.322 DOWNLOADIN file 'myPhoto2.jpg'
06-14-2011 00:08:53.327 UPLOADING file 'C:\Users\vijayu\SyncFolder\myPhoto1.jpg'
```

From the above log snippet it's easy to figure out that a new file myPhoto1.JPG was added, picked up by scanner and it's trying to upload it. Also it's trying to download a different file myPhoto2.JPG (on a different thread) at around the same time. Also this looks like the point of failure as log contains repeated section of the above uploading/downloading sequences. This might be an indication of the application getting into some kind of infinite loop and the issue surfaces only when upload and download happens simultaneously. This can happen when user has added files on both machine1 and machine2 and each is making its way onto the other machine. On the machine where failure is happening, the upload and download are occurring simultaneously resulting in some kind of a failure.

So to summarize we have following information:

- User intent: Addition of file on two machines and syncing to all machines
- Product path: Scanning completed, Simultaneous download and upload task started

Now for playing it back, the playback tool needs to detect these two pieces of information. From the line on scanning along with file name it's straightforward to interpret that the user added a new file myPhoto1.jpg and this condition can easily be reproduced by adding a new file to the path specified.

The lines in log on downloading tasks along with file names indicate that there was newly added file on the other machine that it's trying to download to this machine (where failure occurred). So to simulate this condition, the tool needs to add a file to a different machine to the path specified.

Examining the timestamp for upload and download indicates that these two tasks are happening (nearly) simultaneously. Putting it all together, these are the steps playback tool need to perform

- Create file myPhoto1.jpg in machine1
- Create file myPhoto2.jpg in machine2
- Wait for download to start for file myPhoto2.jpg
- As soon as download starts, also start uploading myPhoto1.jpg

User intent actions can easily be reproduced by adding a new file to the path specified on machine1 and machine2. However controlling the flow to ensure download and upload occurs simultaneously is a little tricky.

In order to accomplish this control over flow of execution our team used the detours[ii] library. In a nutshell the detours library enables hijacking function calls in the product and gives test code a chance to control its execution timings. The way this is used in the example case for file synchronization mentioned above is:

- Create a test binary that detours upload and download functions by providing a detour function
- Load this test binary to the product process

- In the upload detour function wait until download is ready to begin
- In the download detour function wait until upload is ready to begin
- When the condition is met, continue the execution

6. How it can be used for test generation

The test team could use this playback technique to pro-actively identify issues before releasing the product. This can be achieved by taking an existing log for a given scenario (say produced by functional tests), interpreting the log to identify different states in the product and simulating all possible control flow conditions.

For example, in the above file sync case, on creating a file to sync folder we noted that product goes through four states. From this we can come up with the following three cases that can occur simultaneously:

- Download during Scanning
- Download during Uploading
- Download during Realizing

The same playback tool can then be used to simulate the above three conditions and test the products behavior.

7. Conclusion

The log playback has the potential to make reproduction of issues less painful and cheaper. It does require initial investment to come up with the infrastructure for playback, but once it's in place, it can rapidly enable test teams to test different conditions/code paths the product could take and also to reproduce issues reported by the users.

8. Reference

[i] http://en.wikipedia.org/wiki/File_synchronization

[ii] Galen Hunt and Doug Brubacher. *Detours: Binary Interception of Win32 Functions*,

Green Lantern Automation Framework

Sridevi Pinjala

IBM

Twitter: @SriluBalla

Abstract

Once upon a time, the average life for software was 7 years. Some software lasted more than 7 years. Some software lasted less than 7 years. But no matter how long the software lasted, its code was updated, improvised, changed and tweaked frequently. To handle the tweaks and to make sure the other code was not broken, regression testing came into being. Automated tools were invented to handle regression. But most automated tests also needed to be tweaked when the application code was tweaked. When the application interface changed completely, the automated scripts died.

Some companies had several interfaces for one back-end system. They had to pay the price to have all these interfaces tested with the same tests and maintained separately. There was no way out. If one particular interface has hundreds of automated scripts written already, they could not re-use the scripts on other interfaces.

Some companies had frequent changes to the User Interface of the application. Since the automated test case scripts needed to be changed every time the User Interface changed, automation did not add any value nor did it decrease the need for manpower. So, the companies had to rely on manual testing alone.

Many test automation tools came into the market to help out the needy companies. Some tools relied on the objects used for testing, some relied on the User Interface, and some relied on the Document Object Model. Almost all of these tools had a feature - to change the logical name on the elements to give them a unique identification to avoid conflict.

Biography

Sridevi Pinjala is an automation expert at IBM and is now working on IBM test automation tools. She came up with Green Lantern framework when working with Everest Consultants. She is a Certified Scrum Product Owner and currently pursuing MS in Information Management at Aspen University. She considers Job Bach her mentor without whose encouragement, she wouldn't be here to present her ideas. She identifies herself as the American in the Saree. (http://sriluballa.wordpress.com)

1. Introduction

Many companies have invested in test automation tools in hopes that the automation tool will free up manpower by taking over regression testing. The automation tools look at the skins and DOMs of the applications. They are identified by the properties assigned to the elements. (A cell could be identified by the “Row and Column”, a table can be identified by “number”, a link can be identified with ‘inner HTML or text”, and so on.)

When the skin and the Document Object Model of an application changes a lot or little, it becomes unavoidable to update and change the automation scripts a lot or a little too. And if the application User Interface was completely changed (for any reason) the automated scripts will need to be recreated from scratch.

In one of my roles as an automation engineer, I was in charge for creating test scripts for three different interfaces that used the same back end. Their functionality was same, but their user interface was completely different. I started off as everybody - creating scripts for each User Interface. But it felt like too much hard work but like “too little smart work”. So, I needed a framework where I could avoid script death by eighty percent and increase re-usability by eighty percent. I needed a disruptive framework. That is when I came up with “Green Lantern Framework”

Using the renaming feature of the automation tool that relies on the Document Object Model, the test scripts could be created once and used no matter how much the User Interface changed or the even the Document Object Model changed. For example automated test scripts created for Gmail Log-in page can be used for Yahoo Log-in page, Hotmail Log-in page or any log-in page with a user name and password.

I will demonstrate creating automated scripts using Green Lantern framework with the tool Test Complete and scripting language VBScript.

2. Green Lantern Steps

- Choose elements on the page that are targeted for automation.
- Capture the elements in a repository
- Rename them in English as they would appear on the page
- Categorize them per their type (Example – Text Field, Check box, Radio button, etc.)
- Write scripts using the pseudonym names
  - Write a routine for each element
  - Routines for text fields should have data coming from external Data sheets
  - Checkboxes are checked ON or OFF via Data Sheets
  - Log messages, Warning, Errors should be written in the script routine

2.1. Concept –

Main elements in a software application are

- Text fields
- Check boxes
- Radio buttons
- Links
- Menus
- Captions
- Images
- Logos

Every test uses some or all of the elements in a page. These elements are captured and categorized and named as they display. Every test uses some or all of the elements in a page. When these elements are captured and categorized and named the way they would display, it is easy to identify every element, easy to write a script, easy to detect element recognition failures, etc.

2.2. Choose elements for automating login function

Elements essential for Logging into a web page are the URL, the fields - User Name and Password, and button – Sign In.

Figure 1– Choose elements necessary for the test from Gmail Login page

2.3. Capture the elements to the repository

Capture the elements the fields - User Name and Password, and button – Sign In from the Gmail Login page. (Example – Figure 2)

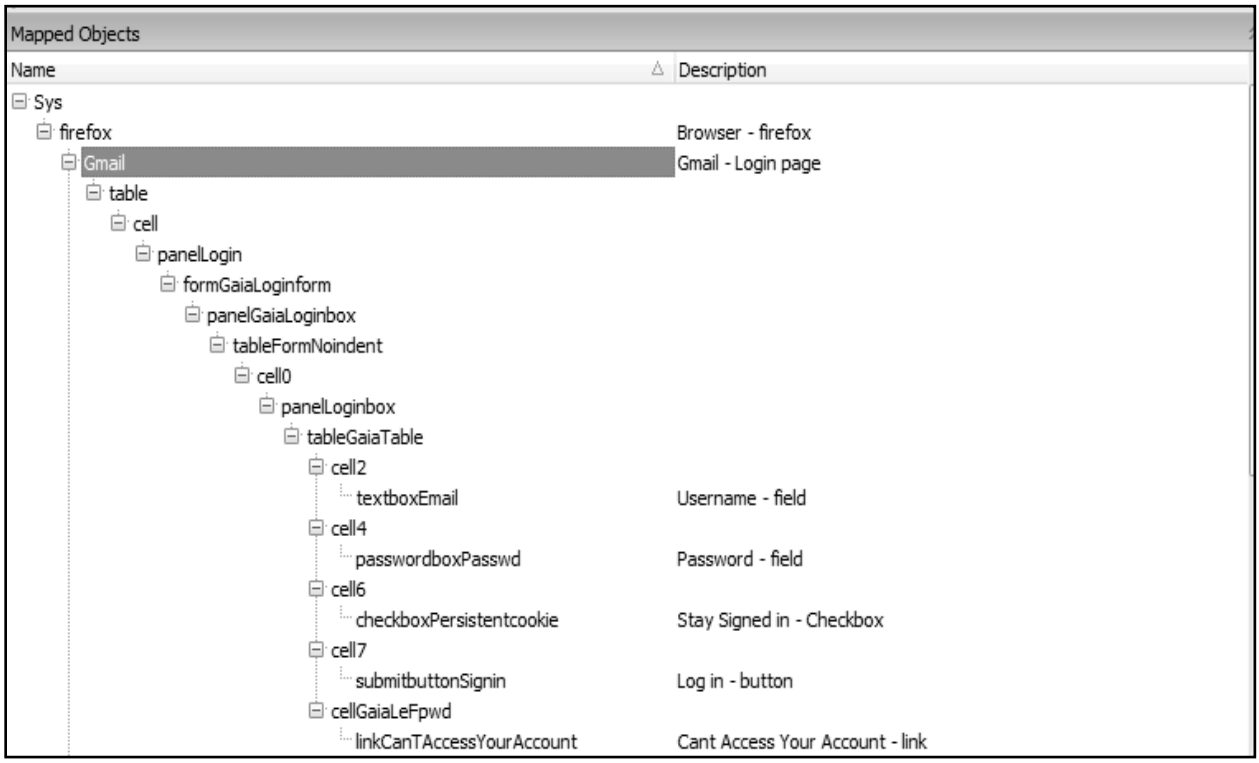


Figure 2 – Gmail Login elements Original

2.4. Rename the logical names of the elements

Firefox = MF

GmailLogin = Gmail_Login

Username (in Gmail) = UserID

Password = Password

Sign In = SignIn

Stay signed in, = StaySignedIn

Can’t access your account? = ForgotPasword

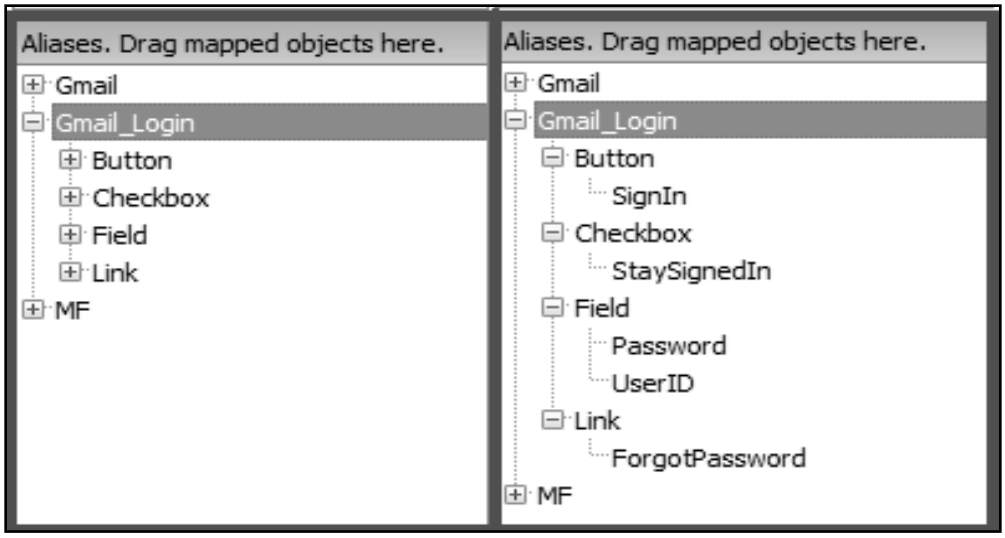


Figure 3 – Gmail Login elements renamed and categorized

2.5. Categorize the elements per their type

To categorize, choose element “panelLoginbox” (which is the parent element for all the field, checkbox, button, link elements). Drag this element and add it under Gmail and rename it as “Field”. Add elements UserID and Password under the Field. Drag “panelLoginbox” again to Gmail and rename it as “Checkbox”. Add element “StaySignedIn” under the Checkbox. Drag “panelLoginbox” again to Gmail and rename it as “Link”. Add element “ForgotPassword” under the Link. Drag “panelLoginbox” again to Gmail and rename it as “Button”. Add element “SignIn” under the Button. The end result with look like Figure 3 (Collapsed on the right and expanded on the left).

2.6. Test Scripts

Using the renamed Aliases elements, it will be effortless and uncomplicated for a new-comer or an experienced programmer to write scripts. (Note - Naming the elements and categorizing them can be decided upon studying the Document Object Model, properties of elements and such.)

2.6.1. Routine for the field User ID

Step1. Start with “`Aliases.`”

Step2. Call the Page or Website you wish to use for the test. In this instance it is Gmail_Login element.

So add – “`Aliases.Gmail.`”

Step3. Call the type of element you want to use for the test. In this instance it is the Field element. So add – "`Aliases.Gmail.Field`."

Step4. Call the element you want to use for the test. In this instance it is the UserID element. So add – "`Aliases.Gmail.Field.UserID`"

Step5. Call the function the step is supposed to perform with the element. A textbox field can either enter "text" or type in "Keys". So add = "`Aliases.Gmail.Field.UserID.Keys`."

Step6. Text or Values to be used could be static or dynamic. I like dynamic. I like to drive my data from the external datasheet. This way any member of the team can come up with the data they would like to use, without ever touching the test scripts. So add = "`Aliases.Gmail.Field.UserID.Keys(D`**`DT`**`.CurrentDriver.Value("UserID"))`"

Step7. I like to perform a few more steps before I input the indented UserID. I like to remove any pre-existing text that might be in the field. For this purpose I copy paste the above code twice and replace the `(D`**`DT`**`.CurrentDriver.Value("UserID"))` with `("^a")` from the first line to select all the existing text. Replace the `(D`**`DT`**`.CurrentDriver.Value("UserID"))` with `("[Del]")` from the second line to delete all the existing text. So copy and paste and modify the lines to -
`Aliases.Gmail.Field.UserID.Keys("^a"); Aliases.Gmail.`
`Field.UserID.Keys("[Del]")`

Step 8. Insert a message in the log to know which UserID was used. So add – `Log.Message("User ID = " + Aliases.Gmail_Login.Field.UserId. Value)`

Step 9. Name the routine to match the element. The end result would look like Figure 4.

```
57  Sub fUserID()
58    Aliases.MF.Gmail_Login.Field.UserID.Keys("^a")
59    Aliases.MF.Gmail_Login.Field.UserID.Keys("[Del]")
60    Aliases.MF.Gmail_Login.Field.UserID.Keys(DDT.CurrentDriver.Value("UserID"))
61    Log.Message("User ID =  " + Aliases.Gmail_Login.Field.UserId.Value)
62  End Sub
```

Figure 4 – Routine for field User ID

2.6.2. Routine for the field Password

Step1. Copy paste the UserID routine and replace the value “UserID” with “Password”. The end result would look like Figure 5.

```
65  Sub fPassword()
66    Aliases.MF.Gmail_Login.Field.Password.Keys("^a")
67    Aliases.MF.Gmail_Login.Field.Password.Keys("[Del]")
68    Aliases.MF.Gmail_Login.Field.Password.Keys(DDT.CurrentDriver.Value("Password"))
69    Log.Message("Password =  " + Aliases.Gmail_Login.Field.Password.Value)
70  End Sub
```

Figure 5 – Routine for field Password

2.6.3. Routine for the button Sign In

Step1. Copy and paste the UserID routine. Remove the 2nd and 3rd and 4th lines.

Step2. Replace the value “Field” with “Button”.

Step3. Replace the value “UserID” with “SignIn”.

Step4. Replace the value “Keys (“^a”) with “Click ()”.

Step5. Insert a message to the log to suggest button Sign In was clicked on.. So add –

```
Log.Message("Clicked button Sign In")
```

Step6. Rename the routine to match the element name and type. The end result would look like Figure 6.

```
76
77  Sub bSignIn()
78    Aliases.MF.Gmail_Login.Button.SignIn.Click()
79    Log.Message("Clicked button Sign In")
80  End Sub
81
```

Figure 6 – Routine for button Sign In

2.6.4. Routine for the checkbox Stay signed In

Step1. Copy and paste the Sign In routine. Remove the 2nd line.

Step2. Replace the value “Button” with “Checkbox”.

Step3. Replace the value “SignIn” with “StaySignedIn”.

Step4. Replace the value “Click() with “Click(false)”.

Step5. Insert a message to the log to suggest the check box Stay Signed In was checked ON. So add – `Log.Message("Uncheck check box Stay Signed In")`

Step6. Rename the routine to match the element name and type. The end result would look like Figure 7.

```
72  Sub cSignedInUncheck()
73    Aliases.MF.Gmail_Login.Checkbox.StaySignedIn.ClickChecked(False)
74    Log.Message("Unchecked check box - Stay Signed In")
75  End Sub
76
```

Figure 7 – Routine for checkbox Stay Signed In

2.6.5. Routine verifying the login

The end result would more or less look like Figure 8.

```
45
46  Sub VerifyLogin()
47      If Aliases.MF.Gmail.Link.Exists Then
48          Call Log.Picture(Aliases.MF.Gmail, "Gmail Invoked", "Gmail Invoked", 300,
     0, -1)
49          Log.Message("Gmail invoked via Mozilla Firefox")
50      Else
51          Call Log.Picture(Aliases.MF.Mozilla_Firefox, "Gmail Not Invoked or
     Recognized", "Gmail Not Invoked or Recognized", 300, 0, -1)
52          Log.Message("Gmail Not Invoked or Recognized")
53      End If
54  End Sub
55
```

Figure 8 – Routine to Verify Login

2.6.6. Routine calling all the login steps

The end result would more or less look like Figure 9.

```
65  Sub LoginSteps()
66     Call fUserID
67     Call fPassword
68     Call cSignedInUncheck
69     Call bLogIn
70     Call VerifyLogin
71  End Sub
```

Figure 9 – Routine to combine all elements for login Function

2.6.7. Main routine for validating the function Login

Many steps, ideas, routines, conditions, go into it. The end result would more or less look like Figure 10 and Figure 11.

```
''' ################## LOGIN SCRIPT FOR Gmail VIA MOZILLA FIREFOX ################
Sub Login()

    Set Driver = DDT.ExcelDriver("C:\SriluBalla\DataSheets\Login.xlsx", "Gmail",
 True)
    Driver.Name = "Login"
    Log.AppendFolder("Gmail LogIn with Browser Mozilla Firefox")

  If Not Aliases.MF.Gmail_Login.Exists Then

       TestedApps.TerminateAll()
       TestedApps.firefox.Run()
```

Figure 10 – Script for Login Routine (continued…)

```
12
13      Aliases.MF.ToURL(DDT.CurrentDriver.Value("URL"))
14      Log.Message("URL =  " + DDT.CurrentDriver.Value("URL"))
15
        If Aliases.MF.Gmail_Login.Exists Then
17         Call LoginSteps
18      Else
19         Call Log.Picture(Aliases.MF.Mozilla_Firefox, "Unknown Error", "Unknown
    Error", 300, 0, -1)
           Call Log.Error("Gmail Log In Not available")
21         Call Runner.JustStop(False)
22      End If
23
         If Not Aliases.MF.Gmail.Exists Then
25          Call Log.Picture(Aliases.MF.Mozilla_Firefox, "Unknown Error", "Unknown
    Error", 300, 0, -1)
            Call Log.Error("Gmail page In Not Available or Log in Failed")
27       End If
28
29     End If
30     Log.PopLogFolder()
31
32     DDT.CloseDriver(Driver.Name)
33
34  End Sub
```

Figure 11 – Script for Login Routine

2.7. Test run results (Log files)

The end result of a test run log would more or less look like Figure 12 and read –

Gmail LogIn with Browser Mozilla Firefox

URL = www.gmail.com (URL text comes from data sheet)

UserID = Srilu.Balla (URL text comes from data sheet)

Password = (URL text comes from data sheet)

Gmail Invoked via Mozilla Firefox

Type	Message	Time	Priority	Has Pict...
	Gmail LogIn with Browser Mozilla Firefox	11:1...	Normal	
	The application "C:\Program Files (x86)\Mozilla Firefox\firefox.exe" started.	11:1...	Normal	
	URL = www.gmail.com	11:1...	Normal	
	UserID = Srilu.Balla	11:1...	Normal	
	Password =	11:1...	Normal	
	The check box is already unchecked.	11:1...	Normal	
	Checkbox Stay Signed In Checked OFF	11:1...	Normal	
	Button Sign In Clicked	11:1...	Normal	
	Gmail Invoked	11:1...	Normal	
	Gmail invoked via Mozilla Firefox	11:1...	Normal	

Figure 12 – Gmail login Test Result

A lot of work went into the Login script. (Phew)

3. User Interface Change

Now imagine the Gmail login User Interface changes from Gmail to Hotmail or yahoo. All the time, effort and ideas put into creating the original script have to be redone in most conventional conditions. But with Green Lantern framework the script is safe as long as there is a repository that matches it.

If the aliases DOM's for Hotmail and Yahoo are prepared similar to the aliases DOM for Gmail, the Scripts will work with little or no modification and log similar messages and check for similar functionality.

3.1. Green lantern Steps for Yahoo Login

Step1. Create Repository for Hotmail Login Page

Step2. Rename the elements and categorize tem by type and rename the logical names to match the aliases names for the Gmail login page. Result would look similar to Figure 13.

Step3. Copy paste the Login Script for Gmail - Open a blank script and name it Yahoo_Login and Copy paste the Gmail Login Script

Step4. Replace "Gmail" with "Yahoo" using 'find and replace' (Ctrl + H)

Step5. Verify the replaced text and run the script. Result would look similar to Figure 14.

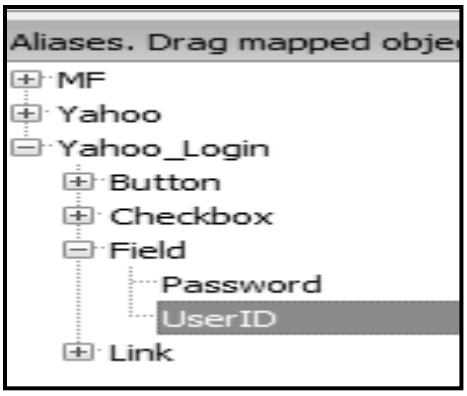


Figure 13 – Yahoo login screen repository

3.2. Test run results (Log files)

The end result of a test run log would more or less look like Figure 14 and read –

Gmail LogIn with Browser Mozilla Firefox

URL = www.mail.yahoo.com (URL text comes from data sheet)

UserID = srilu_kiku (URL text comes from data sheet)

Password = (URL text comes from data sheet)

Yahoo Login successful or Yahoo Login failed.

Type	Message	Time	Priority	Has Pict...
⊟	Yahoo LogIn with Browser Mozilla Firefox	14:4...	Normal	
	The application "C:\Program Files (x86)\Mozilla Firefox\firefox.exe" started.	14:4...	Normal	
	URL = www.Mail.Yahoo.com	14:4...	Normal	
	UserID = srilu_kiku	14:4...	Normal	
	Password =	14:4...	Normal	
	The check box is already unchecked.	14:4...	Normal	
	Checkbox Stay Signed In Checked OFF	14:4...	Normal	
	Button Sign In Clicked	14:4...	Normal	
	Yahoo Login Failed	14:4...	Normal	
	Yahoo Login Failed	14:4...	Normal	
	Unknown Error	14:4...	Normal	
	Yahoo page In Not Available or Log in Failed	14:4...	Normal	

Figure 14 – Yahoo login Test Result

4. A few issues with conventional automation

- Most automation is record and play. This technique will execute a few steps on an application, but wont log full length messages, warnings and won't do comparisons.

- A lot of time goes into automating regression suits. A lot of effort goes into automation. A lot of ideas go into automation. All of this effort goes waste with a few tweaks to the User Interface.

- Several developers (from all over the world) are developing scripts for automation. Unless there was a coding standard defined, each developer would pursue their own style of repository naming and script writing. The automation suits will not be consistent.

- Automated suits with various coding standards are hard to merge and maintain.

- Automated scripts created for one application cannot be used if changes are made to the application. They will need to be updated as well.

- A test had failed due to recognition issues, by looking at the logical names in the repository or the script alone it is not enough to identify which element needs updating. Because sometimes several elements could be named same and several times the names given on the page are not used for the logical name.

5. Conclusion

If this framework is understood correctly, engineers using it can create automation scripts way before the application is developed. Most routines can be utilized over and over after changin a few names in the routines which can be accomplished via find and replace (Ctrl+H).

Several applications with similar functionality could use the same routines and scripts that were already created. Now, the engineer and the testers can come up with more scenarios.

Green Lantern Framework is compatible with the tool TestComplete. Automation tools – Quick Test Professional, Rational Functional Tester, Selenium etc also can have reusable scripts. The approach to these tools varies from the current approach but the concept is similar.

Changes in the DOM and the User Interface of any application is inevitable. During such situations automation script death is inevitable. Many automation tools have a feature to rename the elements being used for tests. When these elements are cleaverly and clearly renamed to suit the elements (also making it easy to remember during script creation), script death can be minimized or altogether avoided and simultaneously the reusability of the script can be increased.

I would like my learned and knowledgeable colleagues to test and question these ideas. I need their invaluable advice, guidance and participation to enhance this framework and to implement on other automation tools.

Parameterized Random Test Data Generation

Bj Rollison
Bj.Rollison@Microsoft.com

Abstract

Testing is the most challenging job in software development. The software tester must select a small number of tests from countless possible tests and perform them within a limited period of time, often with too few resources. Additionally, tests usually employ only a fraction of the possible data that may be used by customers or by malicious users. Whether we are unit testing, testing an API, or executing end-to-end user scenarios or acceptance tests the test data is usually the keystone to many functional tests.

Testers often craft test data representing typical customer inputs, as well as invalid data for a given input control or parameter. But, defining a broad set of test data from all possible inputs for either positive or negative testing is often a non-trivial part of the testing effort. Also, while static test data is useful, the effectiveness of static test data wears out with repeated use in subsequent iterations of a test in which that data is used.

One possible way to increase the breadth of test data coverage is to use random test data. But, random test data is sometimes disregarded because it may not "look like" customer data, or random data may generate false positive results indicating a failure in the system due because of invalid constructs in the random test data. This paper explains the fundamental principles of parameterized random test data generation which can be used to overcome many of the problems associated with random test data. It also demonstrates how parameterized random test data can increase test coverage and expose unexpected issues in software.

Biography

Bj Rollison is a Principal SDET Lead at Microsoft, currently leading a team responsible for testing the foundation services integrating social networking features on the Windows Phone. Bj started his professional career in the computer industry in 1991 building custom hardware solutions for small and medium sized businesses for an OEM company in Japan. In 1994, he joined Microsoft's Windows 95 international team, and later moved into a test management role in the Internet division working on Internet Explorer 3.0 and 4.0 and several web-client products.

As the Director of Test and a Test Architect in Microsoft's Engineering Excellence group Bj has taught thousands of testers and developers around the world. Bj also teaches software testing courses at the University of Washington, and is a frequent speaker at international software testing conference, and is co-author of the book How We Test Software At Microsoft. Bj is currently interested in the application of random test data generation to increase test effectiveness of test designs through variance in test data, and some of his tools can be found at http://www.testingmentor.com.

Introduction

Test data is a keystone of functional tests designed to evaluate specific computational logic, or error handling behavior. Test cases require test data that is representative of the specific input domain for either positive or negative testing. But, unfortunately in many cases the test data commonly used in testing usually represents a relatively small sample of the total population of all possible test data variations for even a single input field.

For example, any non-trivial input variable the number of possible permutations of the data elements in test data is virtually infinite making it impractical to test all possible input variations. For example, a valid computer name for the NETBIOS protocol may be composed of up to 15 of the 82 valid alphanumeric ASCII characters. The number of possible permutations of a NETBIOS name is $82^{15} + 82^{14} + 82^{13}\ldots + 82^{1}$ or a total of 51,586,566,049,662,994,687,009,994,574 possible test data variants.

At 1 test per millisecond it would take over 6000 years to test every possible permutation of allowable inputs for a NETBIOS name. Exhaustive testing is not practical so the tester must decide how to choose a subset of test inputs from the total population of possible inputs that will potentially expose errors or deviations from the expected behavior, and increase confidence that other input values from the same domain would have similar results to the set of inputs used in a given test.

The test data frequently used in tests usually comes from files or databases of static test data, or from direct input via the keyboard. Static test data is usually compiled from various sources including:

- actual customer data
- data that caused errors in the past, or have caused problems in similar situations
- data based on unique or specialized system knowledge
- intuition

Customer data is valuable because it is usually representative of real or real-like customer data. Customer data may come directly from customers, or from domain or business experts. Test data that has exposed problems in the past can reveal problems in other areas, or help verify the robustness of the software in its ability to deal with problematic input values in the right context. Many professional testers may also have specialized knowledge, or understand patterns of attacks that have proven useful in exposing issues in the past in similar context. Even a tester's intuition in defining test data may occasionally reveal previously undiscovered issues.

Static test data provides value in terms of increased confidence and in some cases may improve early defect detection of common data handling errors. However, the effectiveness of static data used over and over tends to diminish in its ability to provide new useful information to the tester. In other words, relying too heavily on static data contributes to the pesticide paradox (Beizer 2009). Once we test the sets of static test data the effectiveness of that data in providing new useful information diminishes in subsequent iterations of the test case in which that data is used.

Randomizing the input variables is one approach to prevent test data from becoming stale or prevent diminishing value gained from reusing static test data. Increasing randomness reduces uncertainty. But, manually choosing a set of possible variables for a given input may not provide a representative sample of the whole population of input variations. Automated random test data generation can effectively improve coverage of the input variations used in a given test. Automated random test data generation can improve the set of test data from all possible variations used to evaluate a program's functional capabilities.

But, simple randomness may not provide an adequate solution to the test data problem in all situations. For example, we cannot simply generate a random number that is composed of 15 numerical digits to test a web form that requires an American Express credit card number. A recklessly generated random number might not satisfy the criteria for a valid American Express card number and result in a false positive outcome of a test.

Automated random test data generation starts with a model of the input variable for the specific domain. Randomly generated test data from a model cannot be predicted exactly. However, we can constrain certain aspects of randomly generated test data by parameterizing the test data attributes. Automatic generation of randomized test data that is a representative sample from all possible permutations is accomplished by selecting specific properties of the modeled test data attributes from equivalent partitions.

Parameterized random test data generation from equivalent partitions is a systematic approach to automated test data generation. By using parameterized equivalent partitions and simple genetic mutation the tester can generate test data that that is an unbiased representative sample from all possible permutations for a given input. Probabilistic random test data can expose unexpected anomalies, significantly reduce redundancy of test data, reduce exposure to risk, and increase overall confidence in test coverage.

1. Modeling Test Data

The key to effective automated random test data generation is the ability to mimic the input domain space. Some input domains are a simple; for example a range of integers between 1 and 100. But, other types of inputs such as credit card numbers are not as simple as a range of sequential values. For complex input domain spaces a model of the input domain helps identify important attributes of input variables. Reliable automated random test data generation is based on a model of the input domain.

1.1 Input Domain Schema

Some input domains such as email addresses, uniform resource locators (URL), and credit card numbers have various constraints in how they are formed. Also some complex data types are composed of 2 or more components combined to formulate a valid (or invalid) input. Modeling the organizational pattern and creating a schema of the data helps to accurately generate random data for complex data. For example, let's define the schema for credit card numbers. Credit card numbers are non-sequential, and not all number permutations are valid credit card numbers. So, we should begin by decomposing the credit card number into its unique components. Figure 1 illustrates the schema for credit card numbers.

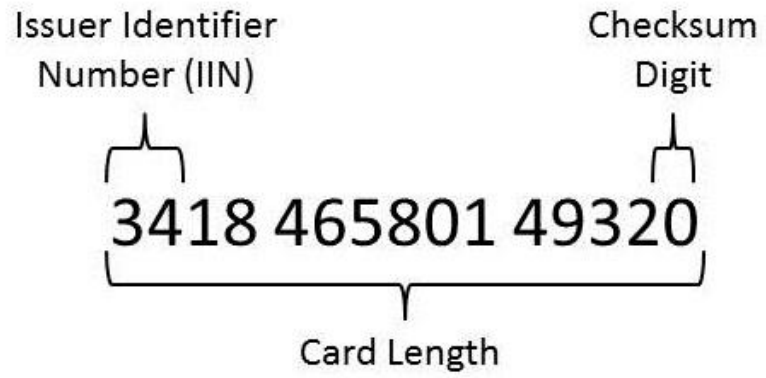


Figure 1. *Credit card number data schema*

The schema for credit card numbers includes:

- Issuer identification number
- Checksum value
- Total number of digits

The first set of digits is the Issuer Identification Number (IIN) used to identify the issuing institution. The IIN values are specific numbers that can be from 1 to 6 digits in length. The last digit of a credit card number is the check sum. All credit cards use the Luhn algorithm (Wikipedia) to validate the credit card number sequence. (The China Union Pay credit card is an exception and the checksum algorithm is unknown at this time.) Finally, credit card numbers can have a length between 12 and 19 digits. For example, Laser, Maestro, Solo, and Switch have card numbers with variable lengths, but, most card issuers use a specific length.

1.2 Data Equivalent Partitioning

Equivalence partitioning based on mathematical set theory has long been used in software testing to categorize input and output variables into equivalent subsets for testing data. The principle assertion of equivalence class partitioning is that any element from a given set has an equal probability of producing the same outcome or behavior as any other element from the same set. Glenford Myers (Myers 1979) stated "…identifying the equivalence classes is largely a heuristic process." This is especially true from a "black box" perspective because the tester doesn't have 'visibility' into the internal structure of the code that parses or manipulates the data.

The heuristics for identifying equivalent partition sets and subsets of input or output variables include:

- **Range** –a linear range of values. For example, integer values from 1 through 100, or upper case ASCII characters A through Z.
- **Unique** – values in a set that may be handled differently. For example, an asterisk character and a forward slash character are both invalid filename characters on the Windows platform, but each character is processed differently by the file I/O APIs. The forward slash will throw an exception resulting in an error message; the asterisk does not throw an exception but changes the state of the listview control in the Save As dialog.
- **Group** – distinctive values in a set. For example, upper case and lower case characters, or the IIN numbers 34 and 37 are a distinctive group of values for the American express card from the larger set of IIN values.
- **Specific** – value that "must be" or "must not be." For example, A password might require at least on upper case character and one number, or some email addresses must not start with a number.

Partitioning data without an explicitly detailed understanding of the algorithm that parses or manipulates the data is not always easy. To be most effective testers should have knowledge of the overall system (hardware, operating system, environment, etc.), the specific domain (client program, programming language used to develop the application, etc.), and the data schema. Limited knowledge or the inability to adequately analyze the data may result in incorrect subsets that ultimately impact the value of randomly generated data from the model of parameterized partitions.

In our example of credit card numbers we should define equivalent partitions of the schema components for each issuing institution. Table 1 illustrates the valid class data partitions for each component of the schema for a small set of credit card types.

Card	IIN	Length	Checksum
American Express	34, 37	15	Luhn
MasterCard	51-55	16	Luhn
Solo	6334, 6767	16, 18, 19	Luhn
VISA	4	16	Luhn

Table 1. *Sample equivalence partitions of some common credit cards*

For each card type in our example, our equivalent partitions include the group of valid values or the value for the IIN, the value or group of values for a valid length, and the specific checksum algorithm. Of course, the only way to validate the equivalence hypothesis is to test the entire population of each set, including all permutations of each number for valid or invalid inputs. But, exhaustive testing is impossible in most situations. So, instead we can generate random samples of the possible set of all data based on our model of that data. Random sampling from the entire data set can increase the breadth of test data coverage, and statistically improve overall confidence that other values in the data set would also produce the same result.

2. Generating Random Test Data

There are numerous random number generation algorithms available in various programming languages. Some programming languages such as Java, Ruby, and C# also have built in class library functions for random number generation. Computer algorithms are actually pseudo-random as compared to “true” random. True randomness is based on some physical phenomenon such as the roll of dice. Pseudo-random generators are initialized with some predetermined value as a starting point such as the system time, and they use mathematical algorithms to select values from a finite set. So, we must ask, are pseudo-random algorithms are reasonably random enough for most generating most types of test data?

The scatter chart in Figure 2 illustrates 5000 random numbers in the range of 0 through 2,147,483,647 using the pseudo random number generator in the System.Random class in the .NET framework. The System.Random class pseudo-random generator uses a subtractive random number generator algorithm introduced by Donald E. Knuth (Knuth 1981).

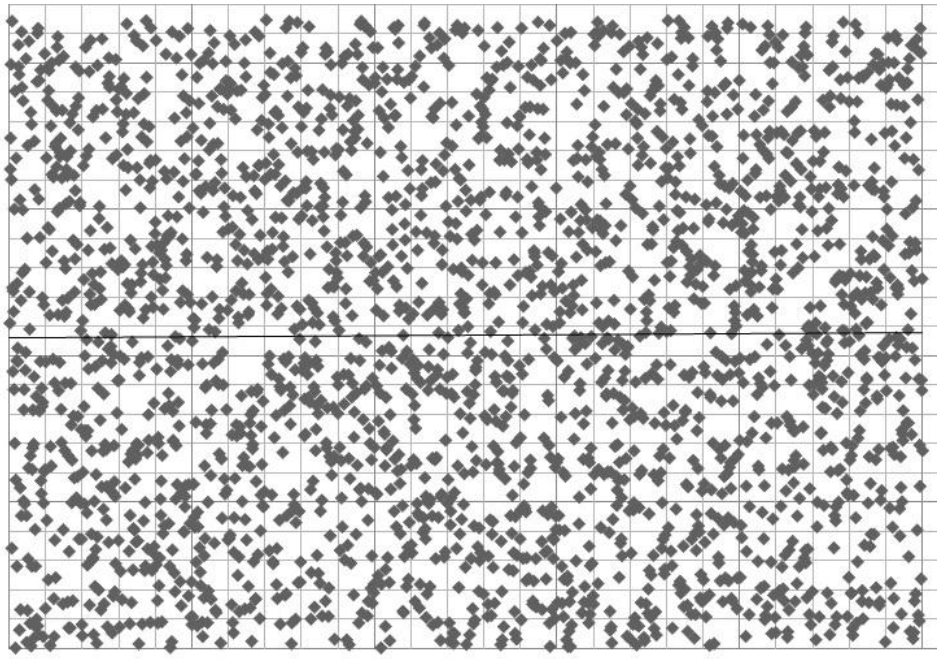


Figure 2. *Scatter chart of 5000 randomly generated integer values*.

The random numbers generated by the System.Random class in Figure 2 appear reasonably random enough for random generation of most common test data. It also appears that this set of 5000 values includes a broad set of data points from the entire set of possible values. Statistically speaking, testing 5000 values out of the total population of 2,147,483,648 would provide a confidence level of greater than 99% with a sampling error of ±3%.

Jeff Atwood (Attwood 2006) compared the 2 pseudo-random generators in the .NET framework; the System.Random class, and the System.Security.Cryptography.RandomNumberGenerator (Attwood 2006). Based on his comparison he also concluded that pseudo-random generators are "perhaps sufficiently random for practical purposes." The Cryptography class pseudo random generator uses multiple environmental factors as compared to the Random class pseudo random generator which uses the system clock. However, we cannot seed the Crypography class pseudo random generator which is critical for random test data generation.

2.1 Seeding Pseudo Random Generators

Software developer and consultant Jonathan Kohl (Kohl 2006) discussed a situation in which his client's excessive use of random data made it virtually impossible to track down failures. A common problem with random data generation is the inability to reconstruct the randomly generated test data in case there is an unexpected outcome. By seeding a pseudo random generator we are able to reproduce randomly generated test data. For example, we can use an integer values as a constructor when we instantiate a new instance of System.Random in C# so we can reliably reproduce randomly generated test data.

Using the same seed value would generate the same random test data each time. So, instead of using a hard-coded seed value, or having to pass a seed value as an argument we can generate a random seed value for each test. The randomly generated seed value is used to seed the pseudo random generator used to generate the test data. The randomly generated seed value should be logged in order to reproduce a random sequence in case of unexpected behavior. The code sample below is one possible implementation of how to instantiate a new instance of Random by either providing a seed value, or by randomly generating a seed value.

```
interface IPseudoRandomGenerator
{
  Random GetPseudoRandomGenerator(out int seedValue);

  Random GetPseudoRandomGenerator(int seedValue);
}

public class PseudoRandomGenerator : IPseudoRandomGenerator
{
  /// <summary>
  /// Generates pseudo random object using a randomly generated seed value
  /// </summary>
  /// <param name="seedValue">Out integer value to log randomly generated
  /// seed value variable for reproducibility of random generated data</param>
  /// <returns>Pseudo random object</returns>
  public Random GetPseudoRandomGenerator(out int seedValue)
  {
    Random randomSeedGenerator = new Random();
    seedValue = randomSeedGenerator.Next();
    return GetSeededPseudoRandomGenerator(seedValue);
  }

  /// <summary>
  /// Generates pseudo random object with a specified seed value
  /// </summary>
  /// <param name="seedValue">Integer value to calculate starting value
```

```
    /// for the pseudo random number generator</param>
    /// <returns>Pseudo random object</returns>
    public Random GetSeededPseudoRandomGenerator(int seedValue)
    {
      Random pseudoRandomObject = new Random(seedValue);
      return pseudoRandomObject;
    }
  }
```

In this example, the GetPseudoRandomGenerator() method instantiates a new instance of a pseudo-random generator and uses the .Next() method to generate a random seed value.It also provides the out parameter so we can log the randomly generated seed value. It then calls the GetSeededPseudoRandomGenerator() method and returns the pseudo-random generator object for use in our test. By directly calling the GetSeededPseudoRandomGenerator() method and passing a given seed value we are able to reproduce randomly generated test data.This approach is an effective way to improve variability of random test data without having to manually seed a random test data generator. It also enables the tester to exactly reproduce specific test data if the randomly generated data exposes an error.

2.2 Generating Random Test Data From Equivalent Partitions

Equivalent partition sets define possible values for use in each component of complex data. Since each possible set of data can be large we can increase variability by randomly selecting values from each equivalent partition. Then by combining randomly selected values from equivalent sets for each component in the data schema we can form random test data based on the model of a specific input domain.

For example, to generate a valid American Express card number a value of 34 or 37 is randomly selected from the equivalent set of valid numbers for an American Express IIN. Next, 13 digits between 1 and 9 are appended to the IIN to satisfy the 15 digit length requirement. Finally, the randomly generated card number must be validated against the Luhn algorithm to produce a valid or invalid card number.

The code sample below provides a simplified implementation for generating a specific credit card based on the creditCardType parameter. This also illustrates how a parameterized value from the larger equivalent set of all valid financial issuing institutions is used to control the random test data to produce a credit card number that begins with either a 34 or a 37 if we pass the appropriate argument for American Express to the creditCardType parameter.

```
  public interface ICreditCardNumberGenerator
  {
    string GetRandomCreditCardNumber(
      Random randomObject, int creditCardType, bool isValidNumber);
  }

  public class CreditCardNumberGenerator
  {
    public string GetRandomCreditCardNumber(
      Random randomObject, int creditCardType, bool isValidNumber)
    {
      // Instantiate class containing credit card data partitions for each card type
      CreditCardDataPartitions cardData = new CreditCardDataPartitions();

      // Get issuer identifier numbers and card lengths by selected card type
      int[][] creditCardData =
        cardData.GetCreditCardPartitionedDataSets(creditCardType);

      // Get a string representing the issuer identifier number if more than 1 IIN
      // for selected card type
```

```
      string issuerIdentiferNumber =
        creditCardData[0][randomObject.Next(cardData[0].Length)];

      // Get the length of the selected credit card type if more than 1 length for the
      // selected card type
      int creditCardLength = creditCardData[1][randomObject.Next(cardData[1].Length)];

      return GenerateCreditCardNumber(
        randomObject, issuerIdentiferNumber, creditCardLength, isValidNumber);
    }

    private string GenerateCreditCardNumber(
      Random randomObject,
      string issuerIdentiferNumber,
      int creditCardLength,
      bool isValidNumber)
    {
      StringBuilder creditCardNumber = new StringBuilder(issuerIdentiferNumber);

      while (creditCardNumber.Length != creditCardLength)
      {
        creditCardNumber.Append(randomObject.Next(9));
      }

      return ValidateNumber(creditCardNumber, isValidNumber);
    }

    private string ValidateNumber(StringBuilder creditCardNumber, bool isValidNumber)
    {
      if (!IsValidLuhnNumber(creditCardNumber) && isValidNumber)
      {
        creditCardNumber = MakeValidCreditCardNumber(creditCardNumber);
      }
      else if (IsValidLuhnNumber(creditCardNumber) && !isValidNumber)
      {
        creditCardNumber = MakeInvalidCreditCardNumber(creditCardNumber);
      }

      return creditCardNumber;
    }
  }
```

In this sample implementation we begin by instantiating the class containing all of our defined equivalent data partitions for credit card numbers. Next we declare a multi-dimensional array to store the equivalent data sets for the IIN numbers and the card number lengths for our selected card type. The pseudo random generator (randomObject) randomly selects an IIN and card length if more than one value are stored in each array. Finally, a call to the GenerateCreditCardNumber() method appends the remaining digits to the IIN number to satisfy the credit card number length argument value. Since this implementation appends a random number to the randomly generated test data, we also need to validate whether or not this credit card number satisfies the isValidNumber parameter.

2.3 Random Test Data Fitness

In the *Encyclopedia of Software Engineering,* Hamlet (Hamlet 1994) points out “Any testing method must rely on an oracle to judge success or failure of each test point.” Reliable oracles are a common problem with random test data generation. Hamlet also notes that random testing is impossible without an effective oracle. So, a method that generates random test data should also ensure that test data satisfies the fitness requirements for a mechanical oracle to be most effective.

In our example, the randomly generated credit card number must pass the Luhn algorithm if the isValidNumber parameter is true. The oracle for our credit card number to validate its fitness is the IsValidLuhnNumber() method. One possible implementation of the Luhn algorithm is illustrated with the code sample below.

```
/// <summary>
/// This method determines whether a number satisfies a Luhn Number
/// </summary>
/// <param name="number">Array of integers used in the number</param>
/// <returns>True if the number satisfies the Mod10 checksum; otherwise
/// false.</returns>
public bool IsValidLuhnNumber(string creditCardNumber)
{
  // Convert string to int array
  int[] number = ConvertStringOfNumbersToIntArray(creditCardNumber);

  int[] temp = new int[number.Length];
  Array.Copy(number, temp, number.Length);

  // The luhn algorithm
  int sum = 0;
  bool checkBit = false;
  for (int i = temp.Length - 1; i >= 0; i--)
  {
    if (checkBit)
    {
      temp[i] *= 2;
      if (temp[i] > 9)
      {
        temp[i] -= 9;
      }
    }

    sum += temp[i];
    checkBit = !checkBit;
  }

  // if the modulus == 0 it is a valid Luhn number; else invalid
  return sum % 10 == 0;
}
```

In this case ‘valid’ implies the card number satisfies the IIN, length, and Luhn algorithm requirements. So, the card number “looks and feels” like a valid number. Each randomly generated credit card number represents one possible number out of the entire population of valid card numbers that can be used by the issuing institution.

For negative testing, or testing for invalid credit cards we would need to add additional parameters. For example, one invalid random credit card number might include an invalid IIN, but the length and checksum requirements are satisfied. Another variation might be a valid IIN for the card type along with a valid length but it checksum digit would not pass the Luhn algorithm. Additional variations for invalid credit card numbers should be tested.

2.4 Random Test Data Fitness

If our desired test data is a valid credit card number and the randomly generated number does not satisfy the requirements for the Luhn algorithm then it calls the MakeValidCreditCardNumber() method. The MakeValidCreditCardNumber() method is an algorithm that mutates the last digit of the randomly

generated credit card number to a valid checksum value to satisfy the Luhn algorithm. Mutation is one way we can further programmatically craft randomly generated test data to contain specific properties.

Mutating test data uses simplified genetic algorithm (GA) principles. A GA is an optimization technique to produce exact (or approximate) solutions to a problem. The key elements of a genetic algorithm include:

- An abstract representation of the population for a given solution
- A fitness function that evaluates the solution for 'correctness'

Basic GA principles can be used to produce or 'evolve' test data that is representative of the population of possible inputs via recombination or mutation. As already demonstrated the tester can use parameterized equivalent partitions to generate an "abstract representation" of test data. But, sometimes we might need to "mutate" test data to satisfy particular requirements; especially for negative or known problematic testing scenarios.

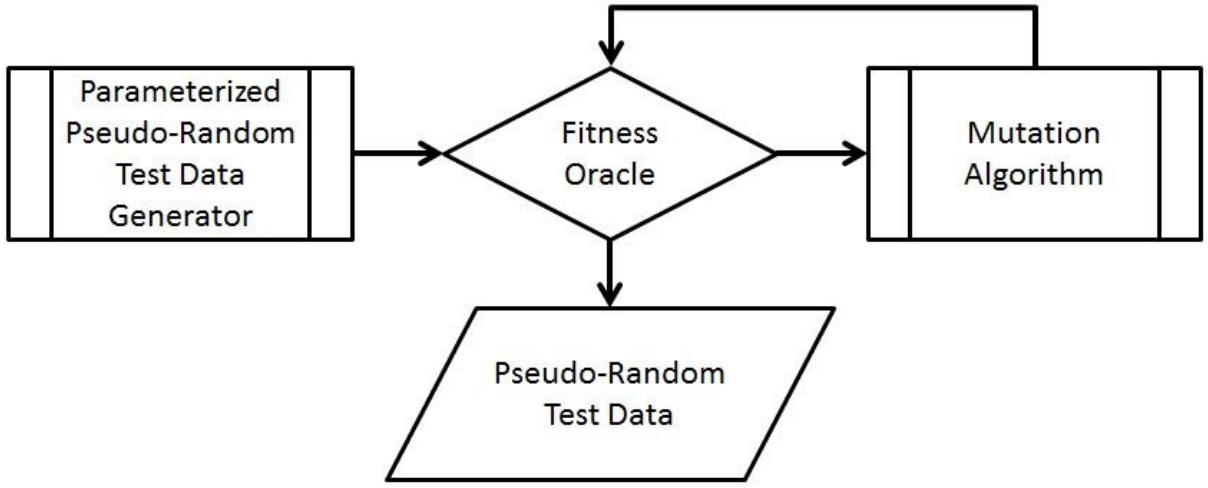


Figure 3. *Simple model of 'genetic' mutation for parameterized random test data generation*

For example let's assume you need a specific permutation of the test data such as a valid checksum value, or you want to generate a random string with a specific Unicode character in a specific location in a string. In these situations parameterized pseudo-random test data generation may not provide the specific outcome. However, we can still use pseudo-random data generation to produce an abstract representation of the data based on the specified properties. The random test data is checked for fitness using a heuristic oracle. If the random test data does not satisfy the desired criteria then the random sequence is passed to a mutation algorithm that mutates the sequence. The mutated test data is passed back to the proto-oracle until the test data satisfies all the criteria for 'correctness' specified by the tester.

This approach uses parameterized equivalence partitions to generate probabilistic random test data, and then use simple 'genetic' engineering principles to mutate the test data (if necessary) until the specified fitness criteria is satisfied for all conditions. Mutating randomly generated test data is especially useful for crafting negative test data variants. For example, we might want to generate a string of random Unicode characters equal to max length, but then replace the last character in the string with a Unicode surrogate pair character which is actually composed of 2 Unicode code point values which may result in a buffer overflow.

3. Pseudo-Random Test Data Generation Library

Pseudo-random test data generators can be used in both automated testing and manual testing. Pseudo-random test data generators are just tools that can help testers increase test data coverage by efficiently generating variable test data that satisfies specific properties. In some cases we might only need one type

of test data such as a random string of characters. But, sometimes we may need test data that resembles a more complete profile of a fictitious person for testing web forms, or sign in to a new social network, a medical records system, or contacts on a phone. In these situations we would need several different types of random test data.

For example, a new contact on your phone might include a name, a phone number, a cell phone number, an address, an email address, a birthdate, notes, and a website URL. Instead of generating random test data for each type of input variable a pseudo-random test data generation library can include a persona abstraction layer. A persona is simply a collection of various types of randomly generated test data.

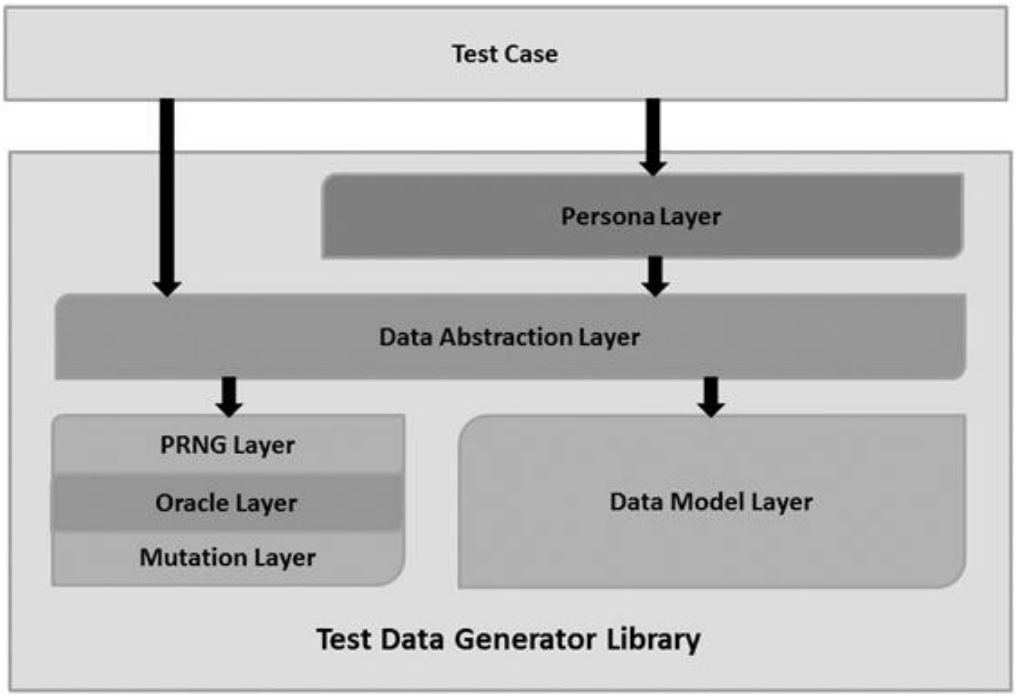


Figure 4. *Architectural layer diagram for a pseudo-random test data generation library*

The Persona layer provides a high level interface that models one or more customer profiles and generates the information or data for that customer profile. For example, a new mobile phone contact may have any or all of the following types of pseudo-random test data:

- Name
- Phone number
- Mobile phone number
- Email address (business)
- Email address (personal)
- Street address
- Website URL
- Birthday (date)
- Anniversary (date)
- Significant other (name)
- Children (names)
- Office location
- Job Title
- Notes
- Ringtone (randomly selected from collection)

The automated test developer can call pre-defined customer personas with any combination of these individual data elements. All these types of test data can be randomly generated and combined in

different patterns to automatically generate any number of new contacts on a mobile device for testing purposes. Of course, different software systems might need different types of information about its customers. Medical records might require social insurance numbers or national identification numbers, and medical insurance numbers.

Occasionally, the automated test developer may not require a complete customer persona, and only require one type of test data such as a date, or a credit card number. The data abstraction layer is the interface for generating various individual types of pseudo-random test data available to the automated test developer and any associated properties that are used to parameterize the attributes of the pseudo-random test data. Table 2 illustrates several attributes that may be found in a pseudo-random Unicode string class in data abstraction layer.

Pseudo-Random Strings
+ Name
+ Address
+ Phone number
+ Date
+ Email address
+ URL
+ String of Unicode characters
+ String 'looks like' sentence
+ Number
+ National identification number
+ Credit card number
+ Driver's license number

Table 2. *Simplified UML class diagram for data abstraction layer.*

The foundation of this pseudo-random test data generation library is the data model layer. The data model layer contains the collections of data partitioned into equivalent subsets for all of the various types of test data. For example, the data model layer contains the collections of all valid issuing identification numbers for credit card institutions. It also contains equivalent partitioned sets of invalid characters in a local email address according to the appropriate RFC and/or various hosting organizations. It may also contain locale specific collections of localized given and surnames.

The pseudo-random number generator (PRNG) layer, the oracle layer, and the mutation layer provide private supporting methods for generating a random object, methods that act as oracles to check the validity of certain types of pseudo-randomly generated test data, and methods to mutate pseudo-randomly generated test data according to specific attributes.

4. Testing with Parameterized Random Test Data

Pseudo-random test data can be used effectively in a variety of situations to increase test data coverage and also expose anomalies that might now otherwise be found with static test data. This section contains examples of issues found using word editor programs to evaluate the effectiveness of parameterized pseudo-random test data generation. This case study used Windows Notepad, and 2 shareware applications; Copywriter and Win32pad. The oracle was a simple string parsing algorithm that compared each byte in the output (the string in each program's rich edit control or the contents of a saved file) with the randomly generated test data applied as input.

4.1 Testing Object Linking and Embedding (OLE)

A common approach to moving data between applications is via OLE, copy and pasting data streams. I used a pseudo-random string generator to automatically generate a string of 1000 Unicode characters

from the Unicode base multilingual plane and set to the Windows clipboard. The string was then copied from the Windows clipboard and pasted it into the rich edit control of each program.

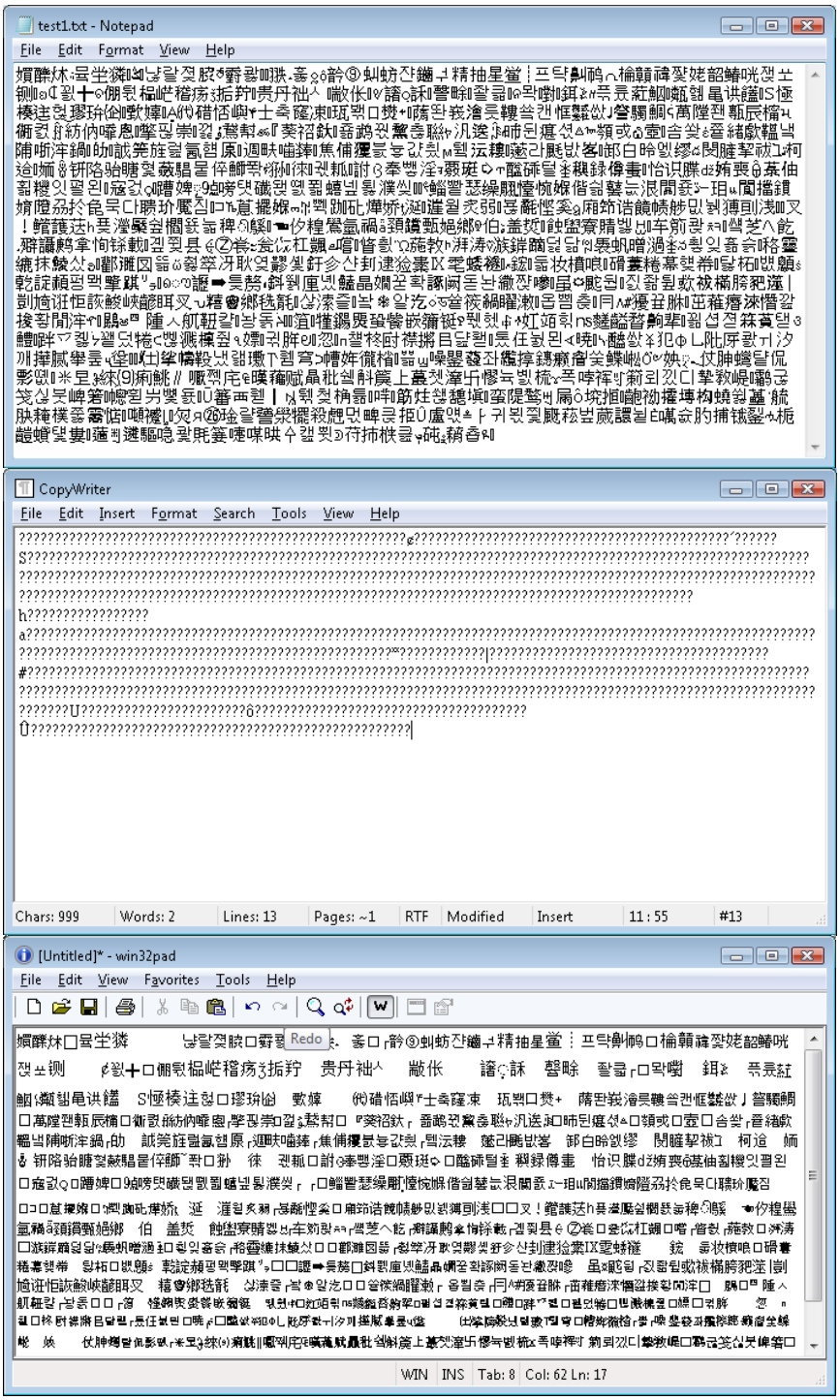


Figure.5: *Results from copy/paste of randomly generated Unicode test data*

As we can see in Figure 2 the randomly generated test data was properly copied from the clipboard into Notepad's rich edit control while the contents of Copywriter's rich edit control was unable to properly handle the incoming stream and converted all characters (except characters in the Unicode Latin 1 range)

to the question mark character (U+003F), which is the Windows default character for unknown character encoding. The Win32pad application visually appears to have preserved the string. Although several characters appear to be missing, a code point is there we simply cannot visually see a glyph. However, when the oracle performs a byte comparison of the string copied from Win32pad the oracle flags an error because all the character code points in the data stream are U+003F; the question mark character. So, even though the random test data might appear to be displayed properly, attempting to save this to a file would result in data loss because the Win32pad converted the stream to the question mark character.

4.2 Testing File I/O Operations

Another common testing scenario involves saving a file, and opening the file. This test saved randomly generated test data to a text file with the Notepad application using the Unicode encoding format. Then the saved file was reopened with the Notepad, Win32pad, and CopyWriter programs.

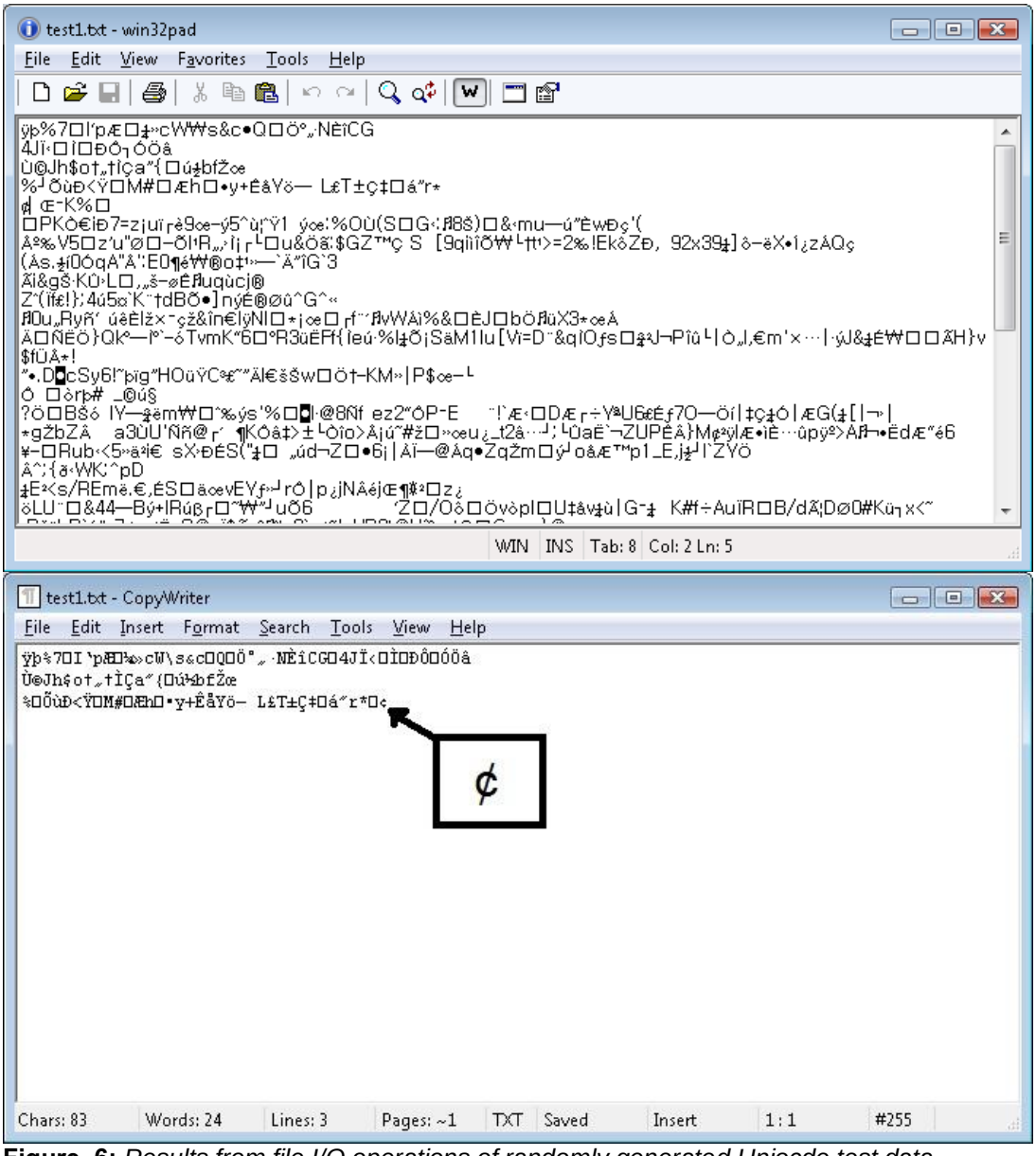


Figure. 6: *Results from file I/O operations of randomly generated Uniocde test data*

Reopening the file in Notepad produced no errors, but the oracle indicated 2 major errors in both the Win32Pad and CopyWriter programs. The most noticeable problem illustrated in Figure 6 is the obvious character corruption. This is caused by the stream reader algorithm for both applications reading the data stream byte by byte instead of by wide_char or as a Unicode encoded stream. However, also notice that the file contents in Copywriter only contains 83 characters instead of 1000 characters that are contained in the file. This occurs because the character immediately following the last character displayed is a Unicode value of U+8CAF. The 8C character code point by itself is the partial line backward control character. Once the stream reader in Copywriter hits a control character it stops reading the rest of the stream which causes the data to be truncated. Interestingly enough, after selecting all the text in the Win32pad application and compare it against the expected result the oracle also indicates truncation after the ¢ character in the stream although additional characters are displayed in the rich edit control.

4.3 Testing With Surrogate Pair Characters

Surrogate pair characters are a uniquely encoded Unicode character composed of a pair of 16-bit code points which make them especially problematic for a developer to parse. Currently, most defined surrogate pair characters are in the CJK language family and the Chinese National Standard GB18030-2000 requires implementation for an official certification of compliance from the People's Republic of China (PRC). Additional surrogate pair characters are defined in American Indian, African, Central Asian, and Ancient language groups as well.

Ad hoc testing of various applications has revealed a variety of problems with randomly generated Unicode strings that contain surrogate pair characters. The majority of errors are unhandled exceptions due to confusion between character counts and byte counts. Other errors include unexpected error dialogs, string or character corruption, and data loss. Random samplings of applications have revealed numerous errors suggesting this is a highly problematic functional problem that requires greater attention.

5. Conclusion

Random test data generation using parameterized data from equivalence partitions is well researched and has been successfully employed in several areas throughout the industry. Thenevod-Fosse, Waeselynck and Y Crouzet (Thenevod-Fosse 1991) presented an approach for random inputs based on specific criteria for improved structural testing. Murphy, Kaiser and Arias (Murphy 2007) discuss a similar approach except their method parameterizes random data according to equivalence partitions for system level black box testing approaches. Also, Demillo and Offutt (Demillo 1991) present a technique for generating test data based on constraints and mutation analysis for fault based testing criteria.

Parameterized random test data is an approach for generating unbiased samples of test data than can be applied to both positive and negative tests. This approach allows the tester to stipulate the specific properties or parameters to generate random test data elements from large populations of possible data, or mutate randomly generated test data in order to satisfy the specified requirements. The ability to generate large sets of both valid and invalid pseudo-random test data for use in testing helps reduce uncertainty by increasing the amount and variability of test data used during the testing cycle.

References

Beizer, Boris. *Software Testing Techniques*. New York, NY: Van Nostrand Reinhold, 2009

Wikipedia, *Luhn Algorithm*, http://en.wikipedia.org/wiki/Luhn_algorithm

Myers, Glenford, *The Art of Software Testing*. New York, NY: John Wiley & Sons, 1979

Knuth, Donald E. Art of Computer Programming, Volume 2: Seminumerical Algorithms, Reading, MA: Addison-Wesley, 1981

Attwood, Jeff. *Computers are Lousy Random Number Generators*, http://www.codinghorror.com/blog/2006/11/computers-are-lousy-random-number-generators.html

Kohl, Jonathan. *Reckless Test Automation*. http://www.kohl.ca/blog/archives/000160.html

Hamlet, Richard. *Encyclopedia of Software Engineering.* New York: Wiley, 1994, s.v. "Random Testing."

P. Thevonod-Fosse, H. Waeselynck and Y Crouzet. "An Experimental Study on Software Structural Testing: Deterministic Versus Random Input Generation," *Proceedings of the Twenty-First International Symposium on Fault-Tolerant Computing*, June 1991, pages 410-417

C. Murphy, G. Kaiser and Marta Arias, "Parameterizing Random Test Data According to Equivalence Classes", *Proceedings of the 2nd International Workshop on Random Testing*, 2007, pages 38 – 41

R. Demillo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Transactions on Software Engineering vol 17, no. 9*, 1991, pages 900-910

B.A. Wickmann. *Some Remarks about Random Testing*. National Physical Laboratory, http://www.npl.co.uk/upload/pdf/random_testing.pdf, May 1998

Testing in Production: Enhancing Development and Test Agility in Sandbox Environment

Xiudong Fei
xiufei@microsoft.com

Sira Rao
sirarao@microsoft.com

Abstract

Developing and testing an application hosted in a sandbox environment presents unique challenges. Development and testing agility is slowed down when validating such applications in production environment. Production environments are complex, have restricted permissions for modifying and using the environment, are costly in terms of deployment and upgrades, and make it difficult for applications to capture relevant logging information and troubleshoot in real-time. To test such applications in production, the problems are aggravated when trying to validate implementation changes, gather information such as logs or if there is a need to troubleshoot the application at runtime as it would require updating the binaries on the back-end or performing in-process debugging by the developer. These problems are also observed when the Test and Ops teams need to deploy test environments to validate the application. Repeatedly setting up test environments is costly and it is hard to simulate a true production environment. All of these constraints slow product development, testing, troubleshooting and thereby delay achieving quality levels needed for release.

This paper introduces a detour concept for an application that is hosted in a sandbox environment. We share the experience of validating the Lync Web application in the context of a Silverlight sandbox environment. In this paper we discuss the tool and framework created to address the above mentioned challenges. We also discuss how we used this tool to intercept Silverlight binaries, modify them and re-direct this to a browser session that hosts the application under test. Additionally, the tool can remotely manipulate objects in the Silverlight application through the use of scripts. With the objects available at hand, we can programmatically manipulate them for out-of-process mode debugging and also use this mechanism as alternate UI automation framework. The tool can be used to enable logging, inflate log size and for product error handling and security testing through fault injection. Using this tool in test and production environments and based on cost of deploying simple test environments, we conservatively estimate a savings of two person-days per month. The paper concludes with lessons learned by our team throughout the development and deployment of this tool, and includes information on practices other teams can implement to achieve similar results.

Biography

Xiudong Fei has been a test engineer at Microsoft Lync group for the last 4+ years. His passion is to create new ways of testing to have business impact, and have fun.

Sira Rao is a Test lead at Microsoft. He has worked on the Unified Communications products at Microsoft for over 7 years. He is passionate about building high quality products that excites customers.

1. INTRODUCTION

We have an opportunity to improve development and testing of applications that are hosted in a Sandbox environment and need to be validated in production environments. Production environments have a high degree of complexity, have restricted permissions for modifying or using the environment, are costly in terms of deployment and upgrades and make it difficult for applications to capture relevant logging information and troubleshoot in real-time. The development and testing agility slows down when validating such applications in a production environment. It is difficult to test such applications in production when trying to validate implementation changes, obtain logs, or to troubleshoot the application at runtime. In addition, product teams must fall back to testing an application in production to help find problems at scale. The scale issues that are usually found in a production environment cannot be duplicated within a test environment. All of these constraints slow product development, testing, and troubleshooting and thereby delay achieving quality levels needed for release.

This paper discusses the tool that we created to address these challenges. We consider the case of our application, Microsoft Lync Web App (LWA), hosted in a Silverlight sandbox environment, and discuss the problems with validating applications in production environments through the introduction of a tool we named SilverlightDetour, that is based on a detour concept. Our Lync Web application needed to be validated for quality in test and more complex production environments and our SilverlightDetour tool allows us to overcome the restrictions of the production environment so as to validate the application under test.

In addition to using the tool, we also actively encouraged “dogfooding” of the Lync Web application and used the tool to inspect and manage the running of our application. In this paper we also provide background on Testing in Production concepts and how they apply to validating applications within a sandbox environment, detail the issues encountered while validating the sandbox environment application and our application in production, and then present a specific project based on a detour concept that addresses the issues related to testing of sandbox environment applications in production environments. We conclude with the lessons we learned, how our solution addresses the issues relating to testing in production and how other teams can benefit directly from our work.

The following terms are used throughout the paper:

- *Dogfooding:* This concept is when product team and others in a company actively use and identify issues in a product prior to release. This practice has been in use in Microsoft since 1988.
- *Microsoft Lync:* The next generation of Microsoft’s unified communications software that enables people to connect in new ways, anytime, anywhere.
- *Microsoft Lync Web App:* This is a real time communications, collaboration client application that is a part of the Microsoft Lync family of products (including server components and clients).
- *Topology:* We refer to this term when we describe the network configuration that consists of various computers and connections between them. In the case of Microsoft Lync Server topology, it consists of the computers running different server roles such as Front end, Edge, Conferencing servers as well as other peripherals such as load balancers, gateways and connections such as switches, WAN links etc. In the paper we will also refer to environments as topologies.
- *Product Package:* The product or application binaries that are retrieved from a back-end system
- *SilverlightDetour:* The tool that is central to the paper, helping to solve issues of testing in production.
- *Remoting:* Usually referring to .NET Remoting, this allows client applications to use objects in other processes on same computer or any other computer reachable over the network or allows for objects in same process to interact with one another across application domains.

2. BACKGROUND

In this section we discuss what it means to test in production, what we consider as sandbox environments, the application and the tools we employ in the context of sandbox environments and testing in test and production topologies.

2.1. What and Why – Testing in Production

Testing in Production ("TiP" in short) uses production environments and topologies to validate a system or application utilizing functional, security, performance and other classes of test. For client teams that develop applications that depend on back-end systems, such as web servers, the application would be validated in test and possibly a pre-production environment before release to production. However, product teams would fall back to testing an application in production to help find problems at scale. The scale issues that are usually found in a production environment cannot be duplicated within a test environment.

2.2. Sandbox Environment

Sandbox environments typically provide a tightly-controlled set of resources for guest programs to run in, such as disk space and memory. These environments are usually closed and restrict the program or application, which is hosted in this environment, from having access to system resources and from what the application can modify. One example of a sandbox environment is the Silverlight application framework.

2.3. Silverlight

Silverlight is a web-browser plug-in and an application framework that enables interactive media experiences and rich business and immersive mobile applications. Silverlight can be hosted within a browser or as part of another stand-alone application. Client applications that are hosted in the Silverlight environment are required to be compiled with Silverlight libraries. Since our client application required browser support, we used a browser to host our application. This meant that the Silverlight applications had to be delivered from a back-end system.

2.4. Product or Client Application

The product or client application we refer to in this paper is Microsoft Lync Web App (LWA). This application is compiled and built for Silverlight and hosted within the Silverlight Sandbox environment. The application has UI and platform components and communicates with back-end server components. To validate LWA application requires that we deploy the server components that LWA depends on directly and indirectly.

2.5. Tools Available for TiP

We needed tools to help with testing in production to overcome some of the obstacles of production environments such as permissions to upgrade, complexity of deployment and difficultly with troubleshooting. There are several tools that enable intercepting network traffic and allow inspection and manipulation of the underlying traffic. These are useful in manipulating HTTP specific traffic, while others can also manipulate lower level protocol traffic. Some of these tools are in the public domain and are well known:

Fiddler – A web debugging proxy which logs all HTTP(s) traffic between a computer and the Internet and allows you to inspect HTTP(s) traffic, set breakpoints, and make changes to incoming or outgoing data.

Middleman – This is an advanced HTTP proxy server with features designed to increase privacy and remove unwanted content.

2.6. Test Topologies

In order to develop and test software and validate it, the application and its back-end must be deployed. Usually the validation consists of unit tests, functional tests, integration tests and end-to-end scenarios. To validate functional, integration, and end-to-end tests we need to deploy test environments that mimic production environments. These test environments or topologies are usually maintained by product teams or a smaller engineering team that services one or more products. In either case, the engineers are forced to balance between deployment and maintenance, and actual product validation or maintaining core engineering services such as test harness, reporting, tracking failures, etc. Additionally, the resources to simulate production environments are not usually available to smaller product or engineering teams, leading to discrepancies between test environments and production environments. This implies that many scenarios would need to be revalidated at a larger scale, on more complex environments, and with multiple simultaneous users.

3. CHALLENGES OF TESTING IN PRODUCTION

3.1. Test environment limitations

Test topologies are usually very simple and use minimal resources. The goal is to validate core functionality and iterate on testing the application and back-end components. The role of the central service engineers is to ensure they provide the product team with simple topologies to validate some functional, integration tests and possibly end-to-end tests and thereby ensure that the product teams have a higher degree of confidence before rolling out the application and back-end changes to the next stage (possibly one or more of the following: “dogfooding” environment, pre-production environment and finally to production). The test topologies usually lack complexity and usage – scale cannot be represented and only simulated to a certain extent, lack integration of advanced components such as load balancers, edge servers. The scale of the test topologies could be enhanced to achieve better approximation of production topologies but the cost would be high for product teams. Additionally, if product team engineers maintain these topologies, they look to minimize their setup and maintenance time even if these machines are virtualized or some install steps are automated.

3.2. Complexity of Production Environments

Maintaining a copy of the production environment has its challenges. There are several quality gates an application and/or the back-end system has to pass through before a production environment can be updated. Installation and un-install require systematic and detailed steps in order to minimize affecting other systems and disruption to users. Additionally, the access and permissions to manage and enable logging of components is restricted. This is deliberate and is needed to ensure that productions systems are not meddled with in ways such as uploading temporary or not fully tested binaries, changing configuration or slowing the system or performance due to excessive logging.

3.2.1. On-Premise (on-prem in short)

Figure 1 shows a sample Production environment that supports high availability and a single data center. Some of the components of this environment include Active Directory, an array of front ends, load balancers, edge servers etc. Other components provide additional functionality in a Lync Server deployment. The Lync Web application depends on many components of the back-end system that are built by other product teams such as conferencing servers, front end and edge servers, PSTN gateways, and load balancers. These are deployed and setup by

other product teams who have expertise on the installation and configuration of these components.

The following topology depicts a Reference Topology for Production Environment: (Source: http://technet.microsoft.com/en-us/library/gg425939.aspx)

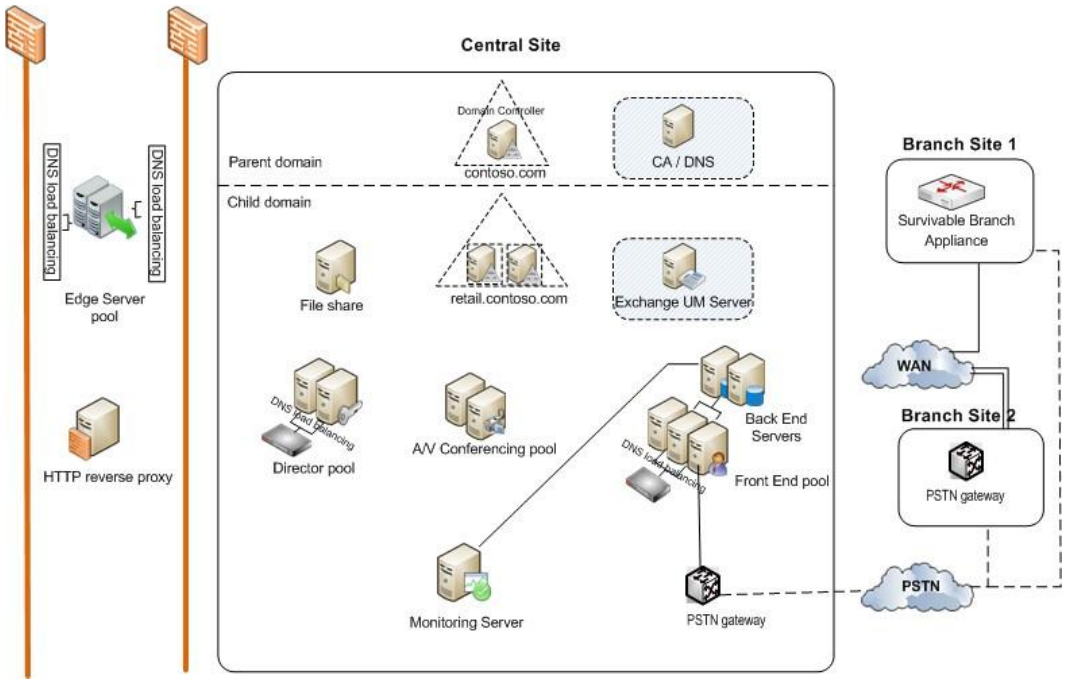


Figure 1 – Sample Topology diagram of Microsoft Lync Server showing the back-end components

3.2.2. Hosted

This type of environment or topology can be deployed by a partner or by the Office-365 service. The hosted or service topologies have more redundancy and scale and typically have more complexity than on-premise deployments, enable more users to use the back-end systems and stricter administrator controls.

3.3. Challenges for Testing in Production

As we discussed earlier, testing in production helps find problems at larger scale and the scale issues that are usually found in production cannot be duplicated within a test environment. However as we noted in the beginning of this paper, production environments come with high degree of complexity, restrict the access to only certain individuals to modify the back-end systems. This makes these topologies costly in terms of deployment and upgrades and makes it difficult for programs or applications (such as client applications) from capturing relevant logging information and troubleshoot in real-time.

3.3.1. Obtaining Logs and Log Size

Usually, the type and level of logging are fixed by default in the product (especially for sandbox environment applications) by the time it is released to production. To investigate and troubleshoot issues, the developer may need more details, so control over logging levels and components are needed. We also identified cases where it was not possible to get the logs if the product hung or exited. In this case, saving the logs to external storage was preferable.

Due to memory size limitations, however, the size of the logs that our application can write is limited. During our testing we observed many issues were missed as the logs saved by application in memory were overwritten due to the fixed size of the logs. In this case transferring the logs to external storage is preferable.

3.3.2. In-process Debugging

The usual debugging approach involves in-process debugging tools such as Visual Studio, which is attached to the specific target process, and the developer sets breakpoints where he or she can inspect and modify the states of variables. This type of debugging is valuable, however it requires stopping the process to inspect variables, see the call stack, etc. Sometimes the developer cannot interrupt the running process (especially when the debugging is timing-dependent) and in this case the only way to inspect the states of variables is through secondary means such as printing variable information into the log file or showing a notification. It is clearly evident that an out-of-process debugging mechanism would be helpful in this scenario.

3.3.3. Silverlight Cross-Domain Policy

Silverlight allows only site-of-origin communication for non-image and non-media requests. In order to enhance security of the backend systems, the server only allows download of the client application from the back-end server. While secure, this restrictive nature of the cross-domain policy reduces developer and test agility as it prevents validating private binaries and fixes. Although one option to overcome this is to run extra scripts to update the cross-domain policy on the back-end, this would not be feasible in production environments.

3.3.4. Validate prior to Deployment

With a production system, it is necessary to pass the quality gates prior to deploying an application or updating any of the back-end components. However, it is helpful sometimes to quickly validate the application in production environments without updating the back-end or even the application that is hosted on the back-end environment. This can provide increased confidence to the product teams having met required validations on production environment prior to updating the application in production.

3.4. Impact to Agility

As mentioned, the challenges of testing in production also directly affect the agility of product development and testing. When “dogfooding”, there is a need to iterate quickly in diagnosing the issue, making the code fixes, and validating the unit tests, functional and integration scenarios without updating the back-end environment or even the client application hosted on the back-end. We need to find and fix issues quickly so that we can encourage and ensure “dogfood” users that their feedback is valued. In certain scenarios, the UI of the application may need to be changed while the underlying platform and back-end server components remain the same. In this case we would like to validate the new UI interactions on the production systems. These changes are not directly possible in production systems due to the restrictive permissions and quality checks that one must go through to deploy new binaries so it does not impact the rest of the system and other users. This implies that product development, testing, “dogfooding”, and troubleshooting slows down the process, requiring more time to achieve desired quality levels.

When testing in production, the install and upgrade process require planning and maintenance, which delays system validation and testing, since deployment times, machine and upgrade issues can frequently delay validation. Ideally the changes made to applications will receive quick feedback from users and the process of fixing and validating can iterate quickly to achieve desired quality levels. Additionally, maintaining these production systems is costly in terms of time and resources.

4. SILVERLIGHTDETOUR SOLUTION

To solve the challenges of TiP, we used a two part strategy. The first was to encourage broad use of the application in production environment by employees outside the product team – which is termed “dogfooding”. This helped increase the development and testing agility, since the product teams iterated quickly on finding, fixing and validating issues that impacted the users in the dogfood group. The second part of our strategy was to create a tool we called SilverlightDetour, that helped us inspect and manage the running of the application in production or test environment, but remained inconspicuous to the users.

4.1. Dogfooding

When team members and other employees use their own product there is frequently a desire to enable better turnaround time between identifying the issue, and providing a fix and re-deploying the updated application. We solved the continuous Dev/Test iteration by encouraging the Dev to use the SilverlightDetour tool to validate their unit and functional tests against both test and production environments. In addition, the test team and central engineering teams pushed back on frequent topology updates and reduced this to monthly updates or when faced with breaking changes. This helped increase dogfooding of the Lync Web application on both test and production environments, and increased the use of the SilverlightDetour tool to inspect and manage the application.

4.2. SilverlightDetour

We needed to validate our Lync Web client application in test environments (simple) and in production environments, which are more complex. The SilverlightDetour tool allowed us to overcome the restrictions of the production environment by redirecting the communication through a proxy.

4.2.1. Overall Architecture

Figure 2 highlights how the SilverlightDetour tool and framework behaves. In the diagram, it shows how the product or application package from the server is intercepted and modified by SilverlightDetour, and the modified package is passed to the user where it is loaded in a browser session. The tool is implemented in C# and uses the .NET framework and runs as a standalone executable (.exe). The tool can be run on a machine that also runs the Lync Web client application, or the tool can be on a remote machine and configured to accept requests from another machine that runs the client application. We established a TCP connection between the tool running on a desktop machine and the client application within the Silverlight sandbox environment. This helped address the issue of collecting logs for and troubleshooting of the application, especially when testing the application in production.

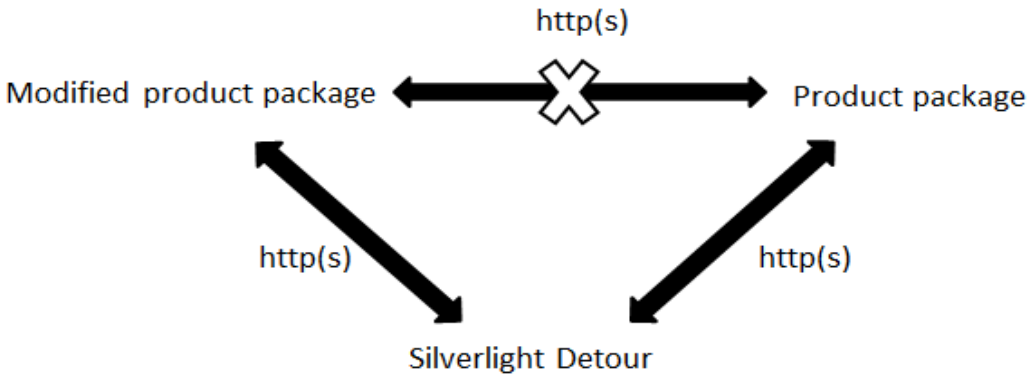


Figure 2 – Overall Architecture

4.3. Implementation Details of the Tool

4.3.1. JavaScript

Using the SilverlightDetour tool we were able to remotely manipulate Silverlight objects through the use of a TCP connection pipe between the Silverlight application and the tool (that is running on a desktop machine). This was done via a small piece of JavaScript that is injected into the HTML page that hosts the Silverlight application. When the client browser loads the Silverlight application, that JavaScript code is executed and creates a Silverlight object from a small piece of XAML. Within this Silverlight object a TCP connection is initiated with SilverlightDetour tool. The code snippet for the JavaScript is mentioned later in Sect. 4.4.7.

4.3.2. JSON

To manipulate objects and identify their type, we needed to serialize and de-serialize them. This was accomplished using a JSON serializer to extract objects to a string format, which is then sent and received using TCP transport. Before an object is sent across the TCP transport, it is serialized and packed into JSON format that can be transmitted across the network. At the other end of the TCP connection, the serialized data is read and is then de-serialized back to the actual object or type that it represents.

4.3.3. TCP connection

As mentioned earlier, a TCP connection is made between the Silverlight sandbox and the desktop machine. The Silverlight object makes a TCP connection to one of Silverlight's allowed ports (4502 - 4532). On this same connection, there are two kinds of data that are transferred: (1) to send and receive the serialized JSON stream and (2) to send logging message back to SilverlightDetour tool. The transfer of JSON stream was used in the manipulation of Silverlight objects and the log messages were used for the purpose of getting logs to the device and inflating the log size.

4.3.4. Use of Fiddler

We incorporated Fiddler's FiddlerCore library into the SilverlightDetour tool. This allowed us to perform HTTP/HTTPS traffic inspection, monitoring and modification with our tool.

4.3.5. Remoting for Silverlight

The SilverlightDetour tool passes objects between Silverlight sandbox and desktop and this is done in a similar manner to .NET Remoting mechanism –behaving as a sort of a .NET Remoting infrastructure for the Silverlight application. For this purpose we included some features of another internal tool that supports limited remoting for Silverlight. We used this remoting to manipulate objects as well as transfer logs from Silverlight environment to a hard disk on a local or remote machine.

4.4. How SilverlightDetour addresses the Problems of TiP

4.4.1. Improves Dogfooding

Since dogfooding was a requirement for most of the Lync family of products, we had to enable non-product team members to use and validate our Lync Web application in production. However, we also wanted to find a way to manage and inspect the application if any issue did come up during testing. One important feature of the tool is the ability to replace product binaries, a feature that was often used by those dogfooding our application. This helped us work with Dev and other Test members to increase adoption of SilverlightDetour tool by highlighting and encouraging use of this feature.

As mentioned in Figure 2, the application package from the server (back-end system) is downloaded via HTTP(s) and is then unzipped. The SilverlightDetour tool modifies the package manifest file to replace any existing assemblies or add any additional ones, re-zips the package and then sends the HTTP(s) stream back to the web browser session. The

browser loads the modified package and now the “most recent” versions of the client application can be validated against the same back-end system such as a production environment. This allowed us to validate code fixes and changes before broad roll-out on production environments.

4.4.2. Avoids Install/Uninstall pains

Since the application binaries can be replaced using the SilverlightDetour tool, the constraints of deployment and upgrading back-end systems and the delays involved in that can be by-passed to a large extent. In addition, the test team and central engineering teams pushed back on frequent topology updates and reduced this to monthly updates or when faced with breaking changes that affected our Lync Web application, such as between platform and server back-end components.

4.4.3. Overcomes Production Environment restrictions

Using the tool we are able to validate the application, obtain logs, troubleshoot application at run-time in production environments – both on-premise and hosted. This helped us avoid the restrictions and permissions relating to getting logs for application, troubleshooting application, replacing application binaries to validate code fixes and waiting on deployment upgrade cycles.

4.4.4. Inflating Log Size and Retrieving Logs

We establish a TCP connection between the tool and client application. Once the connection is established, users can dynamically disable or enable relevant log components, change the level of logging and extend the size of logs in real-time by making the entire disk available for log storage, rather than what’s available in memory and limited by the application. Using the SilverlightDetour tool we pass objects between the Silverlight sandbox and the machine running the tool as depicted in Figure 3. Retrieving logs is similar to inflating log size where we pass objects and transfer logs from sandbox to local machine.

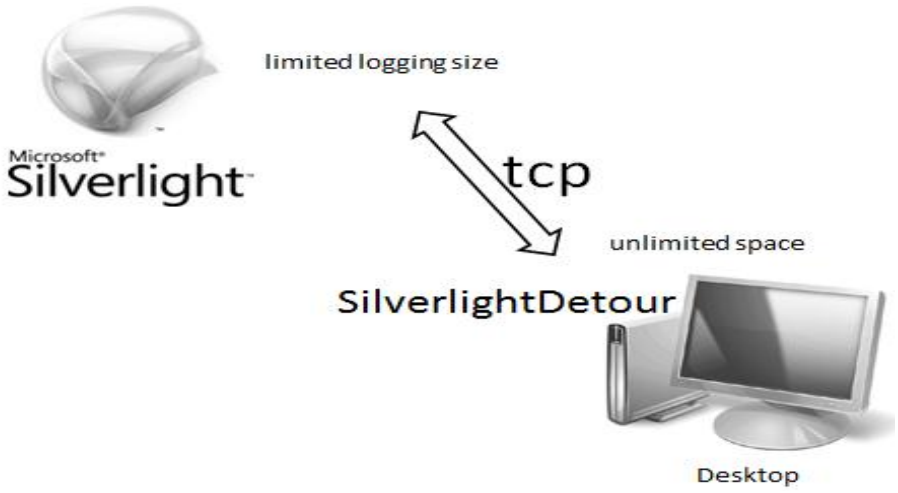


Figure 3 – High level design of Unlimited Silverlight Application logging

4.4.5. Debugging made easy

With the power of being able to pass objects between the Silverlight sandbox and desktop, it enabled useful and interesting scenarios for the product team. The communication and object passing mechanism allowed the application to pass objects by reference remotely between the Silverlight environment and the tool and we were able to inspect and modify the values of variables, the state of Silverlight objects, and to validate the behavior while the application is running. This allowed us to enable debugging on these objects by remotely controlling the

Silverlight UI elements and enabled out-of-process debugging during runtime without attaching to a debugger. In addition, we could also set breakpoints based on HTTP(s) request / response, which is hard in traditional debuggers.

In this example the SilverlightDetour tool manipulates the remote Silverlight objects in our client Silverlight Lync Web application. The figure and code snippet shows an example of out-of-process debugging using scripts such as PowerShell. The tool obtains a Silverlight object having type as text box (referred to as *displayName*), modifies the text property, inspects the value of '*TextAlignment*' property and modifies to right alignment. We can also use this tool to debug based on HTTP(s) request / response sent between client and back-end.

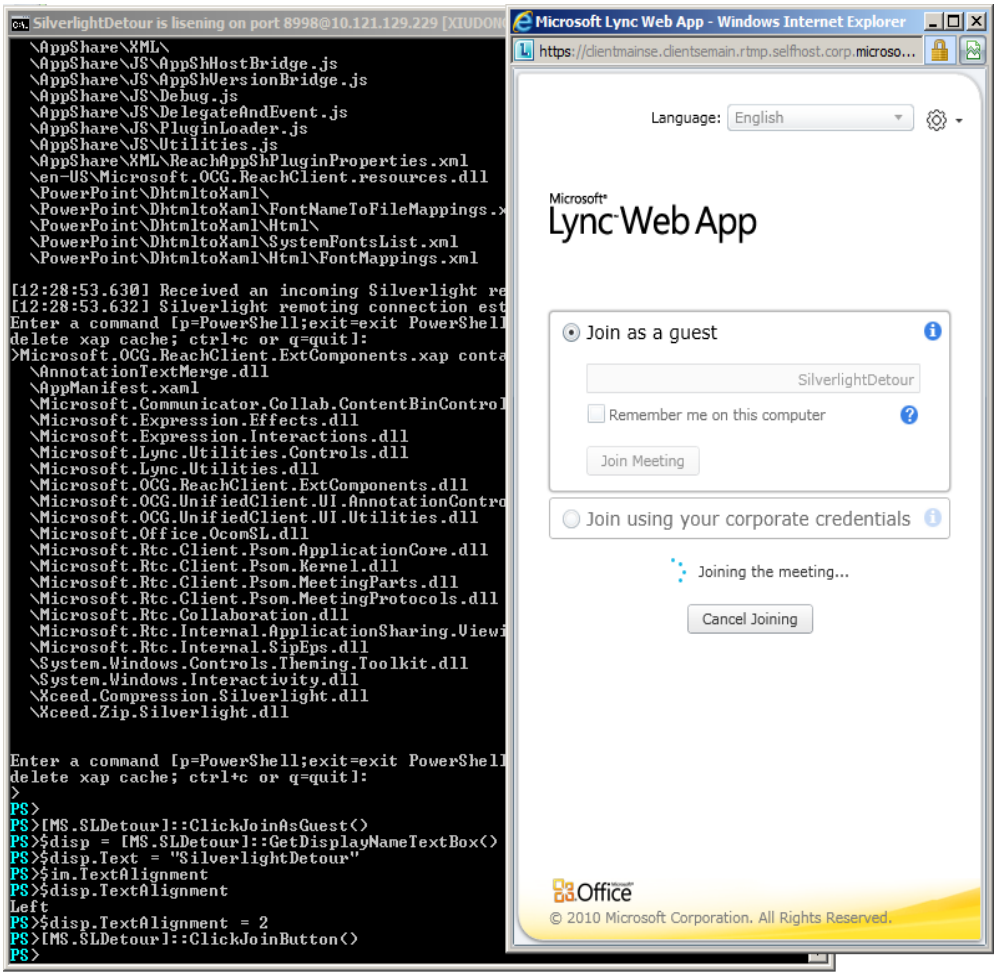


Figure 4 – Using Scripts (PowerShell) to control remote Silverlight objects

PS>$disp = [MS.SLDetour]::GetDisplayNameTextBox()
PS>$disp.Text = "SilverlightDetour"
PS>$disp.TextAlignment
Left
PS>disp.TextAlightment = 2

4.4.6. Facilitates Remote Assistance applications

Our tool also supports a lightweight form of remote assistance. If a customer needs help on using a feature of the product, the support team can start a TCP listener and accept the

incoming request from the customer. The support team can issue commands to highlight and modify the Silverlight UI elements and hence guide the customer in a simple and step-by-step manner. This also allows the support team to switch to debugging and inspecting the public and internal variables of the application should anything go wrong. This seamless move from Helping to Debugging is possible within a single tool, while it would be cumbersome to set up and use a traditional debugging tool just for the latter case.

4.4.7. Allows Programmatic Manipulation via Scripts

Silverlight does not normally allow remote manipulation of objects. To overcome this limitation, we had the idea of injecting JavaScript during the interaction between the client and back-end. The tool checks if a HTML page contains the Silverlight application by searching for 'object' node which contains type="application/x-silverlight-2". The tool then injects the following: "param node name = onload , value = CreateBridgeServiceLoader"

```
1.  <object "type="application/x-silverlight-2">
2.      <param name="source" value="product.xap" />
3.      . . .
4.      <param name="onload" value="CreateBridgeServiceLoader"/>
5.  </object>
```

Here the code snippet shows how to use JavaScript to create a Silverlight object and that object initiates a TCP connection to a certain machine (e.g. machinename.domain.com) and port (e.g. "4510")

```
1.  <script type="text/javascript">
2.      function CreateBridgeServiceLoader(sender, args) {
3.          var url = "tcp://machinename.domain.com:4510";
4.          var silverlightPlugin = null;
5.          silverlightPlugin = sender.getHost();
6.
7.          if (silverlightPlugin != null) {
8.              try {
9.                  var obj = silverlightPlugin.content.CreateFromXaml("<host:BridgeServiceLoader
    xmlns='http://schemas.microsoft.com/winfx/2006/xaml/presentation'
    xmlns:x='http://schemas.microsoft.com/winfx/2006/xaml' xmlns:host='clr-
    namespace:Microsoft.RemoteBridge.Host;assembly=RemoteBridge.Host'
    ProxyServerAddress='" + url + "' />");

10.                 if (obj == null) {
11.                     alert("Failed to call CreateFromXaml to establish a Silverlight bridge
    connection.");
12.                 }
13.             }
14.             catch (exception) {
15.                 alert("Failed to create BridgeServiceLoader.");
16.             }
17.         }
18.         else {
19.             alert("Page does not contain Silverlight application.");
20.         }
21.     }
22. </script>
```

Figure 5 shows a sequence diagram detailing the interactions between the SilverlightDetour tool and the Web Browser and the Server (Back-end system).

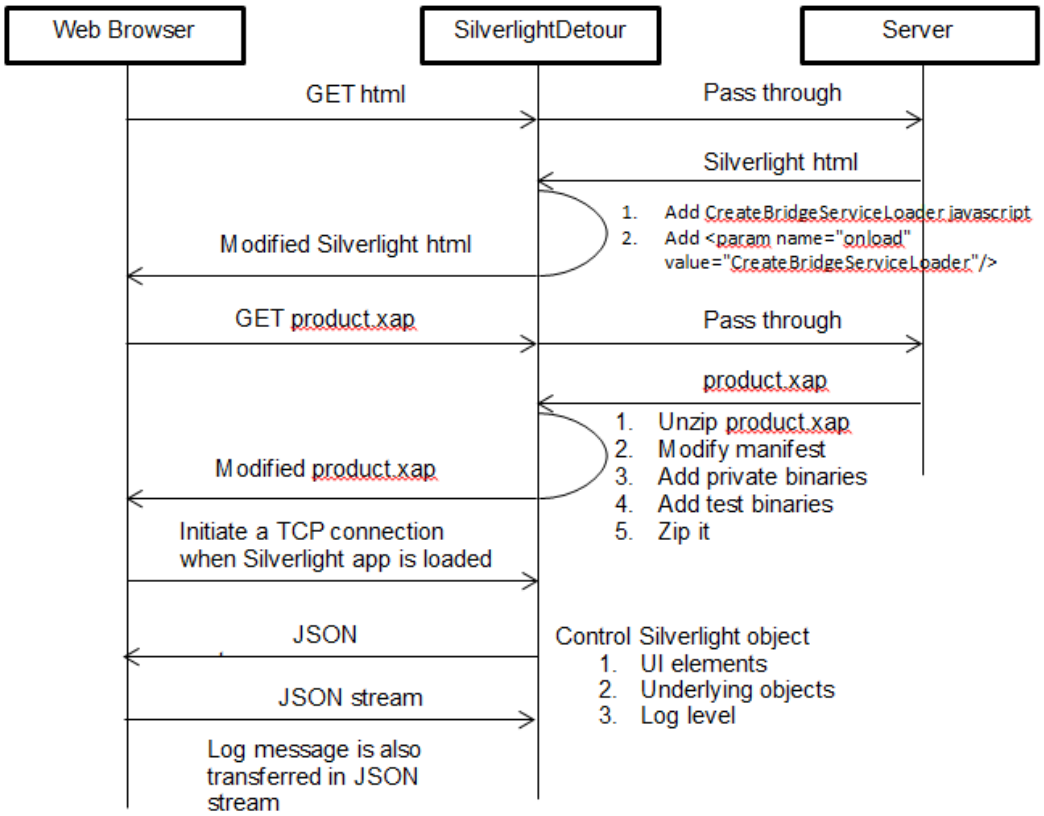


Figure 5 - Sequence Diagram showing actions between web browser, SilverlightDetour tool and server (back-end system)

4.4.8. Provides Alternate UI Automation Mechanism

Using the TCP connection between the Silverlight sandbox and Desktop and the ability to manipulate Silverlight objects, it is easy to manipulate remote Silverlight UI elements using Silverlight Class library. Figure 4 in sect. 4.4.5 shows an example of UI element modification using the tool.

Code snippet:

```
FrameworkElement rootElement =
Application.Current.RootVisual.Cast<FrameworkElement>();

static public TextBox GetTextBox(string name) {
        return rootElement.FindChildren().ByName(name).Cast<TextBox>();
        }

static public RadioButton GetRadioButton(string name) {
        return rootElement.FindChildren().ByName(name).Cast<RadioButton>();
        }

static public void ClickRadioButton(string buttonName)  {
        RadioButton button = GetRadioButton(buttonName);
        ToggleButtonAutomationPeer peer = new ToggleButtonAutomationPeer(button);
        IToggleProvider provider =
peer.GetPattern(PatternInterface.Toggle).Cast<IToggleProvider>();
```

```
        if (provider != null)
            provider.Toggle();
         }

    static public void ClickJoinButton()  {
            ClickButton("JoinButton");
            }
```

4.4.9. Bypasses Cross-Domain Policy

As mentioned in Sect. 3.3.3, to enhance security of the back-end, the server allows only downloading the client application from the server. This means that validation of private binaries against production systems would not be possible unless the policy is bypassed. SilverlightDetour is used as a cross domain policy server so that the tool returns a lower restriction policy when the Silverlight client requests for the cross-domain policy ("GET clientaccesspolicy.xml"). Hence the tool overcomes the limitations of the Silverlight sandbox environment while continuing to provide agility to the Dev/Test cycle. Below is an example of the policy the tool provides:

```
<?xml version="1.0" encoding="utf-8" ?>
<access-policy>
  <cross-domain-access>
    <policy>
      <allow-from http-request-headers="*">
        <domain uri="*" />
        <domain uri="https://*" />
        <domain uri="http://*" />
      </allow-from>
      <grant-to>
        <resource path="/" include-subpaths="true" />
      </grant-to>
    </policy>
  </cross-domain-access>
</access-policy>
```

4.4.10. Enables Fault Injection and Security testing

Given that we are able to manipulate the Silverlight objects as well as perform out-of-process debugging by tracking the HTTP/HTTPS requests or responses, we were able to manipulate error codes, change protocol schema elements, simulate hard to obtain errors and perform fuzz testing of the XML parser in interesting ways. However, for lack of time we covered only the mainline scenarios. An additional complication to using the SilverlightDetour tool was that our Lync Web application and back-end needed to support compression so that the LWA would work well on low bandwidth networks. We were able to test with compression as well as using the SilverlightDetour tool to negotiate not having compression. This was done by changing a success response to a compression request to an error thereby disabling compression for the duration of the browser session.

5. RESULTS

5.1. Dogfooding

We observed that the SilverlightDetour tool helped the product Dev team to resolve issues before code check-in by running unit and functional tests against test and production environments. The test team also used the tool to run functional and integration tests to identify issues. This

reduced the Dev/Test loop in terms of code check-in, testing, resolving issues, and verifying fixes. In addition, when the product team encouraged dogfooding by other individuals, issues were quickly debugged through the use of the tool and once again brought to the product team's attention.

5.2. Appreciation and Collaboration

The tool and framework that was developed by the product team members received a lot of positive feedback across diverse teams (Redmond, India development/test teams). In addition, we also received Peer Recognition awards for the benefit to the product and our partners in the development team.

As the SilverlightDetour tool gained more usage within Dev /Test teams, there were more suggestions and requests to add features and provide extensibility hooks. This required managing the requests in a similar manner as the requests for adding additional features to a product. This was resolved by having additional members contribute to the enhancement of the tool and by balancing priorities of tool development and product validation among the other product team members.

5.3. Improvements in Testing and Validation

The SilverLightDetour tool allowed for improved testing of the Lync Web Application in terms of functionality, stability, overall correctness of the application and of the back-end environment.

1) Ability to obtain logs when the application crashes, exits abruptly, or hangs due to application, network, or back-end issues
2) The tool allows teams, that use Silverlight, to validate the application package that is downloaded from back-end system
3) Ability to simulate errors and verify user experience related to scenario and error message
4) Improved testing of resiliency scenarios, which were not previously possible since we could not bring down the production systems to test error and fail-over scenarios

5.4. Cost Reductions and Productivity Gains

5.4.1. Dev/Test spend minimal time on Setup

By testing in production, we found the Dev team could stop spending time investing on topologies to test their client side functionality. Testing in production also benefited the product team test engineers as well as the central service engineering team since they spent less time setting up and updating the topology, dealing with topology setup failures and also communicating with admins / engineers on production environments.

Prior to using this tool, the development cost was one person-day per month, while the test team required two person-days per month. After using the tool these costs went down, and there was an increased use of the tool to test changes and bug fixes to the application against a Production environment before upgrading the back-end. If we were to consider deploying a topology that resembles production topology (as mentioned in sect. 3.2.1), the costs would go further up and would require a dedicated resource for managing the machines, deployment of the many server roles.

5.4.2. Affects Productivity in other tasks

With the cost savings obtained from spending less time on setup, installation and maintenance of topologies, the product team engineers (including the authors) were able to spend the additional time validating the quality of the product, automating more tests, and coordinating better testing through a wider set of users in the dogfood group. The overall productivity of the Dev team went up as they spent little or no time worrying about topology and the productivity of the Test team went up as they spent less time on routine and difficult tasks such as setup, deployment and maintenance of the topology.

5.4.3. Gains in Production systems

This cost cannot be estimated by us since it involves setting up things in an automated manner as well as running manual configuration changes and requiring one or more dedicated resources. We do estimate that by providing the client application to other individuals, we identified more issues at scale. For example, in large scale online meeting scenarios we observed sluggishness and high memory usage by the client application. We had fixes for these within a week and used the tool to verify the behavior in another large scale online meeting on production environment before rolling out the updates to production environment.

5.4.3.1. *On-Premise*

We primarily validated our Lync Web client application using the SilverlightDetour tool against this type of environment since many customers have the Lync Server deployed on their premises.

5.4.3.2. *Hosted*

We validated our application using the tool against hosted environments towards the latter part of the release. The same gains and costs are applicable here as these have more complexity than on-premise production environments.

5.4.4. Agility

Agility increased as the Dev / Test accelerated the rate of validating unit and functional tests, and troubleshooting and debugging issues. The product teams were quickly able to validate the application in production environments without updating the back-end or the application that is hosted on the back-end environment. This increased the quality of the application and the product was able to go out to the next gate (either for dogfooding or Production) having met a higher quality bar.

5.4.5. Benefits due to Remote Assistance and Debugging

The remote assistance and debugging features of the SilverLightDetour tool were demonstrated and used by the team members in a number of ways. One such instance was when we wanted to debug on a Mac machine and show how this can be used as a help mechanism. Having the Mac machine point to the computer running the SilverlightDetour Tool, we were able to perform basic troubleshooting on that computer. In fact, this tool can be used along with any of the major browsers that are supported on Windows OS and Mac OS. This type of run time debugging can help improve productivity of the product team (Dev or Test).

5.5. Lessons Learned

5.5.1. Getting people to use the tool / framework

With any new tool or framework the hurdle to cross is adoption of the new way to work. In this case, the product test team began using this tool and pushing back on topology upgrades. Simultaneously, we worked with the product Dev team to use the tool to validate changes before code check-in. After highlighting the benefits and seeing the tool in action, the product Dev team adopted the tool and gradually began evangelizing it among other Dev and test members involved in developing UI and back-end system for the products.

Eventually the tool became vital to development for validating their unit and functional tests, debugging remote issues, validating in production environments with minimal effort and improving the development to test to release cycle.

With the Dev team evangelizing the tool, the entire product team was more careful with checking in breaking changes that needed fixing or updating the tool and framework. This changed the perception of how Dev / Test teams worked so that we were working to mutually benefit the other and in turn help ourselves. The balance shifted more towards finding ways to utilize the tool to validate product quality and less towards routine server and topology maintenance and deployment.

5.5.2. Paid more attention to the network traffic

This tool and framework helped us understand the details of how our product worked at the protocol and network level. Product Dev / Test members became more aware of the working as well as understood underlying issues. This helped us improve quality of the product through fault injection means and manipulation of the UI – as detailed in sect. 4.4.10

5.6. Challenges Faced

Although the framework supports testing in production for the majority of the scenarios, there were some challenges that the product team faced with this framework. In these cases, the product team had to spend some time to upgrade/install a new version of the test topology, validate the quality and then work with admins and engineers to upgrade the dogfood, end-to-end, or production environments.

The challenges were mainly the protocol schema mismatch and timer changes to match sending / receiving data between the client application and the back-end environment. These changes forced the product team engineers to update the test topology as well as ensure that these changes are timed to occur before the next roll-out of the product (application and back-end) to the next stage viz. dogfooding and/or Production. While it would have been possible to "exclude" the breaking changes and compile the application to work against the older versions of the back-end (that reside in production environments), we found that this caused more confusion and had the negative effect of not validating the product correctly.

5.7. Implication to Test In Production

As customers move to having their back-end infrastructure hosted online (in the cloud), the need to validate at scale and in complex environments is a necessity. Deploying such environments is not possible by product team engineers or the smaller service engineering teams. Hence the rapid iteration of the client applications (in sandbox) will need to be validated against Production environments without updating the back-end systems or even the hosting environment for the application. This will require finding solutions to validate client application changes at scale and getting enough "dogfooding" of the product before actual production roll-out. Framework and Tools such as our tool will help accelerate the product development, testing, "dogfooding" and iterate to achieve the quality levels needed before release.

5.8. How Other Teams can Use this

We looked at other teams that employ client applications delivered through the server and observed the class of Web and plugin based applications. Although the tool and framework can be extended to pure web based applications, we primarily focused on applications that are dependent on a plugin such as Silverlight. Other teams / products that utilize such plugins can employ the tool / framework in their specific environments to validate their applications in production. In addition, these products can utilize the different aspects of the framework for purposes of troubleshooting issues, developing programmatic manipulation of the UI such as for UI automation or to test security by fault injection.

In demonstrations within the company, we have seen interest in exploring the benefits and extensibility of the framework. We state that there are no product team decisions on whether they plan to utilize this framework. However, we see benefit in utilizing our framework especially in client server environments such as Bing Maps and other Silverlight based products. In such applications we need to keep the back-end Production environment up and try multiple instances of the client application to validate functionality, user experience while ensuring access to vast databases, supporting multiple users, minimizing disruption to large systems and reducing communication around upgrades / maintenance.

6. CONCLUSIONS

We had initially set out to make our client application testable in order to improve the quality of the product, but found that there were bigger problems to solve. As our focus changed we observed that we could implement a framework that accelerated the product development and testing of applications in sandbox environment for Production environments. In addition we observed that we could iterate rapidly in terms of identifying issues, troubleshooting in real-time and validating fixes all in production environments. This allowed testing client application at scale and in complex environments while minimizing the need for product team engineers to be involved in setup or deployment of even simple topologies. Having client applications running in production or near production environments provided increased feedback to product teams through "dogfood" programs and in turn provided the users and customers with higher quality products via faster development and testing cycles.

In addition our tool had the interesting side effect in terms of providing additional range of testing techniques and use – such as using it for fault injection in security testing, programmatic manipulation using scripts that can be used as alternate mechanism for UI automation and providing for remote assistance either as a troubleshooting or as a help mechanism. Given the current business environment where there is a need to iterate rapidly in developing client applications and testing these applications in complex production environments that are either on customer premises or hosted online in the cloud, there is a need for tools and framework such as ours.

We expect to continue investigating ways to further improve this tool and enhance its functionality. In addition we will look at how this tool and its variations can be adopted by other product teams as well as how other product teams employ or implement other mechanisms to develop and test client applications for the Silverlight sandbox environment and other sandbox environments, the cost savings and productivity enhancements that can be realized and how these applications can be rapidly iterated upon within production environments.

REFERENCES

Wikipedia. "Sandbox (computer security)," http://en.wikipedia.org/wiki/Sandbox_(computer_security) (accessed August, 2011)
Wikipedia. "Eating your own dog food," http://en.wikipedia.org/wiki/Dogfooding (accessed August, 2011)
Microsoft. "Silverlight.NET," http://silverlight.net (accessed August, 2011)
Microsoft. "Making a Service Available Across Domain Boundaries," http://msdn.microsoft.com/en-us/library/cc197955(v=VS.95).aspx (accessed August, 2011)
Microsoft. ".NET Framework Class Library for Silverlight," http://msdn.microsoft.com/en-us/library/cc838194(v=vs.95).aspx (accessed August, 2011)
Microsoft. "Lync 2010," http://office.microsoft.com/en-us/lync/ (accessed August, 2011)
Microsoft. "Welcome to Microsoft Lync Web App," http://office.microsoft.com/en-us/communicator-help/welcome-to-microsoft-lync-web-app-HA101908015.aspx (accessed August, 2011)
Microsoft. "Reference Topology With High Availability and a Single Data Center," http://technet.microsoft.com/en-us/library/gg425939.aspx (accessed August, 2011)
Fiddler. "Introducing Fiddler," http://www.fiddler2.com/fiddler2/; http://www.fiddler2.com/Fiddler/Core/ (accessed August, 2011)
Middleman. "Middleman filtering proxy server," http://middle-man.sourceforge.net/ (accessed August, 2011)
Microsoft. ".NET Remoting Overview," http://msdn.microsoft.com/en-us/library/kwdt6w2k(v=VS.71).aspx (accessed August, 2011)
JSON. "Introducing JSON," http://json.org/ (accessed August, 2011)
Wikipedia. "Fuzz Testing," http://en.wikipedia.org/wiki/Fuzz_testing (accessed August, 2011)

Application Monitor: Putting Games into Crowd-sourced Testing

Vivek Venkatachalam (vivekven@microsoft.com)
Marcelo Nery dos Santos (marsant@microsoft.com)
Harry Emil (harryem@microsoft.com)

Abstract

Software test teams around the world are grappling with the problem of testing increasingly complex software with smaller budgets and tighter deadlines. In this tough environment, crowdsourced testing can (and does) play a critical role in the overall test mission of delivering a quality product to the customer.

At Microsoft, we firmly believe in using internal crowd sourced[1] testing to help us reach high levels of test and scenario coverage. To achieve this goal, we use the dogfood program where employees volunteer to use pre-release versions of our products and give us their feedback on a regular basis. However, for a volunteer based crowdsourced testing effort to be really effective, one needs the ability to direct the crowd to exercise certain scenarios more than others and the ability to adjust this mix on demand. What if one could devise a mechanism that provides the right incentives for the crowd to adopt the desired behaviors ime?

This paper describes how we conceived of, designed and implemented Application Monitor, a tool that runs on a user's machine and allows us to detect usage patterns of Microsoft Lync[2] (the software product that this team of authors worked on) in near real-time. The paper then describes a simple game we incorporated into the tool with the goal of making it fun for the crowd. The game also provided us with the ability to direct their efforts to test high-risk features, by appropriately changing incentives.

One learning point was that even crowd behaviors that attempted to game the system served the ultimate purpose, which was to increase testing of specific scenarios. The paper will discuss this and other takeaways as well as point out key issues that other teams wishing to start similar efforts should consider.

Biography

Vivek Venkatachalam is a software test lead on the Microsoft Lync team. He joined Microsoft in 2003 and worked on the Messenger Server test team before moving to the Lync client team in 2007. He is passionate about working on innovative techniques to tackle software testing problems

Marcelo Nery dos Santos is a software engineer on the Microsoft Lync test team. As a remote employee for 15 months, he was an active dogfood user of the product himself. He has also worked on tools to help test performance for the Lync product. His main interests are working on innovative tools and approaches for enabling more efficient testing.

Harry Emil has worked at Microsoft on the Windows and Office testing teams since 1989. His research focuses on the wisdom of crowds, workplace productivity, and real-time multilingual communications.

[1] Crowd sourcing is the process of harnessing the wisdom of crowds to achieve tasks.
[2] Microsoft Lync is an enterprise solution for communications, delivering on Instant Messaging, Voice, Conferencing and more. For more information, please visit: http://lync.microsoft.com

1. Introduction

Wikipedia defines crowdsourcing as “the act of outsourcing tasks, traditionally performed by an employee or contractor, to an undefined, large group of people or community (a "crowd"), through an open call.” (1) It allows organizations (both for-profit and non-profit) to tap into the collective intelligence and labor of a set of people external to the organizations, to accomplish tasks much more efficiently than if they had to be performed in house. As a matter of fact, Wikipedia itself is a shining example of the power of crowdsourcing. Crowd sourced testing, which is simply the application of the crowd sourcing concept to the domain of software testing, is a useful technique to consider to solve a common problem that all software test teams face; i.e. the problem of verifying the quality of their software under a multitude of different environments. Crowd sourced testing implies that at least some portion of the testing is performed via a loosely co-ordinated approach across a crowd of diverse testers that are not officially part of the test team. This allows the test team to focus their limited resources on verifying software quality under a very well defined and constrained set of conditions (that reflect the common case) while still ensuring test coverage across the broader set of hardware and software configurations that exists out in the real world relatively cheaply and efficiently by leveraging the power of the crowd.

Productivity games are another phenomenon becoming increasingly popular in the business world. The central idea is to introduce creative and fun game-like elements into an employee’s regular workflow, to make work seem more like playing a game than just completing a set of routine tasks. The aim of this approach is to increase employee motivation and morale and thereby indirectly increasing their productivity. A great example of a productivity game is the Language Quality Game. (2)

In this paper, we describe our effort at combining these two key ideas in a series of experiments to increase the test coverage of Microsoft Lync during the last few months before release-to-manufacturing. Microsoft Lync is an enterprise communication and collaboration system that offers users a rich set of features (instant messaging, user presence, voice/video calls and conferencing, content sharing and many others). It comprises of a set of server components (collectively known as the Lync server topology) and a set of clients that run on multiple platforms (Windows, Mac, Web client amongst others) and connect to and communicate with the server components to provide this rich functionality. The authors were on the team responsible for verifying the performance aspects of the flagship Lync client, the client running on the Windows platform ; all future references to Lync in this paper imply this specific Lync client.

Since the underlying technologies were based on work we had done for automating our performance test system, we start the paper by briefly explaining the performance test system, to help the reader get a basic understanding of these technologies. In the subsequent section, we describe how this system we built to collect performance data in a lab environment was modified and extended to allow us to also automatically collect performance data from our dogfood users, i.e. our crowd of testers. Next, we describe how we added games into the mix and expanded the scope of the project from a mechanism that collected performance data passively from users to a mechanism that actively tracked scenarios that users were running and provided them with incentives to run a desired set of scenarios. We then conclude the paper by presenting some key takeaways from this project and ideas for improvements.

2. Background

Two of the authors (Vivek and Marcelo) had been primarily involved in designing and implementing a performance test system for the Lync client. We will now give a brief overview of the process to set the stage and provide readers with the necessary background to follow the paper more easily:

a) First, performance scenarios were identified along with defining elapsed time goals for each of these scenarios. Examples of scenarios identified are SignIn, SignOut, and MakeAudioCall etc.

b) Next, for each scenario, for example, SignIn, instrumentation was added to Lync source code by adding a SigninStart event in the piece of code which handled the click of the SignIn button and by adding a SigninStop event in the piece of code that executed when the user had been successfully signed in. The instrumentation was done by using APIs provided by the Event Tracing for Windows (ETW) toolkit[3].

c) The SignIn scenario was then automated using UI automation. To accomplish this, we wrote some test code that, when run, would simulate a real user signing in by entering her credentials and then clicking the SignIn button.

d) The Windows ETW system would be used to capture the SigninStart and SigninStop events in a binary EventTraceLog (ETL)[4] file. This file contains a log of events (with detailed description) along with timestamps that indicated when the events occurred.

e) An analysis process would use the Windows ETW API to parse the log, read individual events in the log, match start and stop events for a given scenario appropriately and compute the elapsed time for the scenario by computing the difference in the time stamps between these events. The average of computed results from a number of the reasons was used as the estimate for what response time a user might reasonably perceive when exercising that scenario.

3. Introducing crowdsourced testing to the process

Almost all products that are built at Microsoft go through an internal crowd sourced testing process called dogfooding (2), which is basically the idea that employees eat their own dogfood before it is released to the external world. This has long been a key part of the test strategy at Microsoft since it allows product teams to get broad test coverage for real user scenarios against a multitude of hardware and software configurations. Participants in the dogfood program for a product are typically comprised of volunteers from across the company that are interested in trying out the cutting-edge version of the product and want to help the product team improve the quality of that product. The Microsoft Lync team too had its own set of dogfood users that would use the product in their regular work and report issues back to the product team for further investigation. While this worked well for issues related to the functionality of the product, we really did not have a good way to get objective performance data back from these users.

As described in the previous section, we already had a working automated performance test system to track the performance for the Lync client in a controlled lab setting. While we had the ability to collect performance data in the lab, we didn't really know how well the performance measured in the lab would translate to the actual performance of the product observed by our internal crowd of testers i.e. the community of Lync dog food users. After putting some thought into ways to collect performance data from dogfood users, we realized that we could modify and extend key pieces in the performance test system and use these to build a tool that could get real-time elapsed data from a dogfood user's machine. The two pieces different from a lab setting are indicated below:

a) Instead of UI automation driving a scenario, it would be a real dogfood user exercising the Lync UI.

[3] A system that provides application programmers the ability to start and stop event tracing sessions, instrument an application to provide trace events, and consume trace events. http://msdn.microsoft.com/en-us/library/bb968803(v=VS.85).aspx

[4] The binary log file generated by an ETW tracing session.

b) Instead of the resultant ETW events being written to a file to be processed after the fact, these events would be processed in real time by another executable program that had registered with the ETW system to in order to receive real time ETW events from the Lync process.

To achieve this, we built a tool that could run alongside Lync, consume ETW events from Lync in real time, detect when a pair of matching start and stop events had been received and output the computed elapsed time in its UI. In addition, the tool reported the computed time back to a central service that we maintained. Since the tool was designed to be easily modified to consume and process events from any Windows application, not just Lync, the tool was given a generic name: Application Monitor. The tool was designed to run unobtrusively in the Windows Notification Area[5] and no additional intervention was required of the user once they had installed and running. It also had an auto-update feature built in so we could push out new features and bug fixes without requiring any additional work from the user other than the initial step of installing it.

Application Monitor was made as a separate tool because we wanted it to be completely isolated from the actual product code. That way, the product code would have instrumentation added only for the performance markers itself, but the consumption and processing of the fired events would be made by a separate process. By using the Windows ETW infrastructure, this task would be achieved by placing minimal impact on the product actual performance and we would also leverage the fact that when no listeners are subscribed to the events, the ETW infrastructure will actually ignore those events with no performance impact.

Once Application Monitor was running, it automatically collected performance data for the scenarios that users were executing assuming the Lync source code for those scenarios had been instrumented. Coupled with a mechanism to send this data back to a central location, we now had an effective way to track elapsed times for scenarios not just in a controlled lab environment but also on real user's machines. The only thing we needed was to convince dogfood users to install Application Monitor separately from the Lync client and leave it running. We found that this simple requirement to install a separate application was an obstacle to get widespread deployment across the dogfood user population, a topic we address in some more detail in the concluding section.

4. Adding games to the mix

The fact that we had this ability to track elapsed times for scenarios meant we also now had even more basic and potentially more valuable ability: the ability to track Lync scenarios being executed on dogfood user's machines. In effect we now had a lightweight, near-real-time telemetry system that let us know which scenarios were being executed the most, which were being executed the least and which had never been executed (based on the sample of users that chose to install Application Monitor). During our internal evangelization effort (to get people running Application Monitor and provide us with real performance data), it caught the attention of a set of people on our team (our co-author Harry was among this set) that had spent considerable time in designing, developing and deploying productivity games internally at Microsoft. After some discussions, all of us realized that the telemetry functionality in Application Monitor could be used to power games related to dogfooding Lync. This was when we decided to add game elements into Application Monitor to a) incentivize users to participate more extensively in the dogfood program and b) add a bit of fun to the dogfood process.

Besides greater dogfood adoption, the game elements also provided incentives for users to install Application Monitor on their machines since the game required Application Monitor to be running, thereby increasing the quantity of performance data that we got from user's machine. Based on this set of discussions and some creative design and coding work, we implemented and deployed a set of

[5] This is more popularly known as the System Tray or SysTray, the lower right section of the Windows task bar.

experimental games based on Application Monitor functionality. A brief summary of these games are presented in the subsequent subsections.

4.1. Easter Eggs

Easter egg games were engaging images that popped up on the dogfood user's screen when they completed a certain set of scenarios in Lync. The scenarios had to be exercised in a specific order and would indicate that the dogfood user had unlocked an achievement. These encouraged ad-hoc exploration of Lync features (and hence more test coverage for the product) as players tried to uncover these Easter eggs and unlock achievements. To make things even more interesting, creative folks from the team came up with what were affectionately known as "HaiClues", clues to discover the steps needed to unlock achievements in the form of Haiku[6].

Some examples of HaiClues can be seen on Table 1 – HaiClues:

Explicit steps needed to unlock an achievement	**HaiClue version**
1. Switch from sharing a PowerPoint deck to sharing a whiteboard 2. Switch from sharing a whiteboard to sharing a PowerPoint deck	We share, we succeed Slides, whiteboard then double back Accomplishment; joy
1. Search for a colleague 2. Add them as a contact in an existing contact group. 3. Expand that contact group	Find a Coworker Groups hold Contacts for Later Growing groups expand

Table 1 – HaiClues

Figure 1 shows an example of an easter egg that is simply a static html page with some text and images.

[6] Haiku is a short form of Japanese poetry consisting of 3 lines of 5, 7 and 5 syllables.

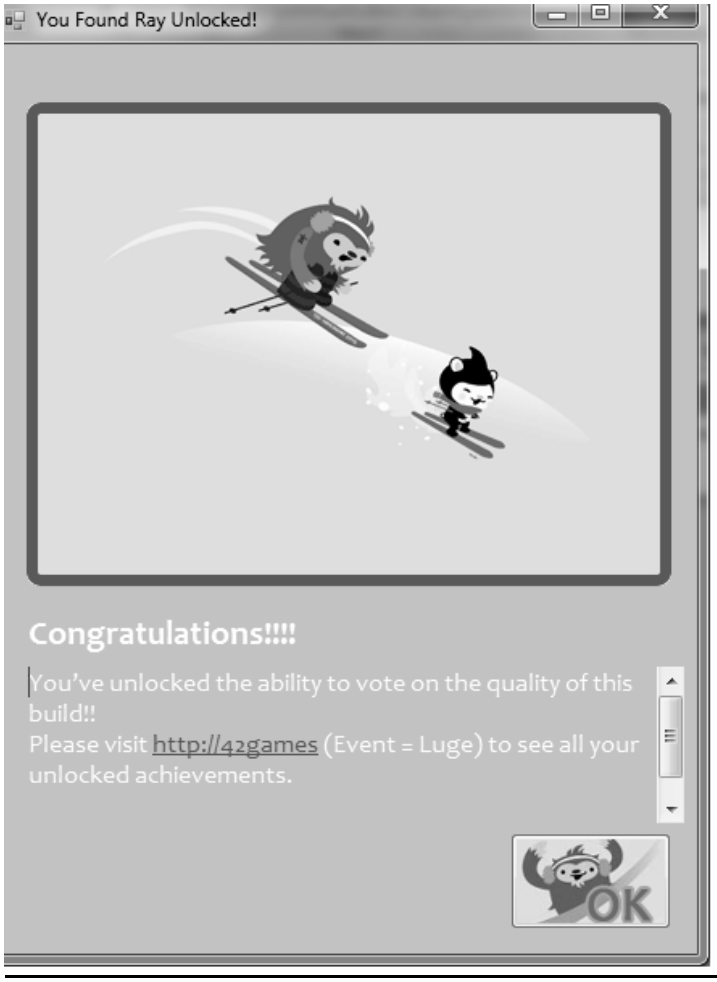


Figure 1 - Example of a static Easter egg

Figure 2 shows an example of an interactive Easter egg in the form of a contact card built on Lync Silverlight controls. In this example, the Easter egg brings up the contact card of a researcher at Microsoft with whom the user can directly communicate using different modalities (click the envelope to send mail, click the text bubble to send IM, click the phone to make a call etc).

Figure 2- Example of a Silverlight based interactive Easter egg

4.2. Olympics themed game

The next step was to make the game more realistic by launching an Olympics themed dogfood game to make it fun for dogfood users to participate as well as use leaderboards as incentive to drive user behavior. Participants self-selected into teams of their choosing and participated in various “events” with the aim being to collect as many achievements as possible in each event, for themselves as well as for their teams. Figure 3 is a screenshot that describes the various ”events” on the Olympic Games, the first two of which used Application Monitor to detect that users had unlocked achievements by completing specific scenarios.

Figure 3- Overview of the games

As an example, the aim of the "Curling" event was simply to try various instrumented scenarios in Lync with Application Monitor running in the background. For each scenario that a user completed and was recognized by Application Monitor, they would get one step closer towards unlocking the corresponding achievement. The number of times the scenario needed to be run to unlock these achievements was set differently for each scenario. ("LaunchCommunicator"[7] was worth 120 points while "ResumeFromHibernate" was worth 5400 points) to motivate participants to run more of the scenarios that mattered to the test team. Since these weights were set on the back-end, we could easily change the weights as needed to get optimal level of scenario coverage by the dogfood users. For example, if we had recently made a big code check-in in the SignIn feature, we could get the crowd of testers to try out this feature more heavily by simply increasing the number of points gained by unlocking this achievement. Some screenshots from various tabs of the leaderboard for the "Curling" event are shown below as an illustrative example (names have been removed to protect privacy of game participants).

[7] Microsoft Office Communicator was the former name of Microsoft Lync

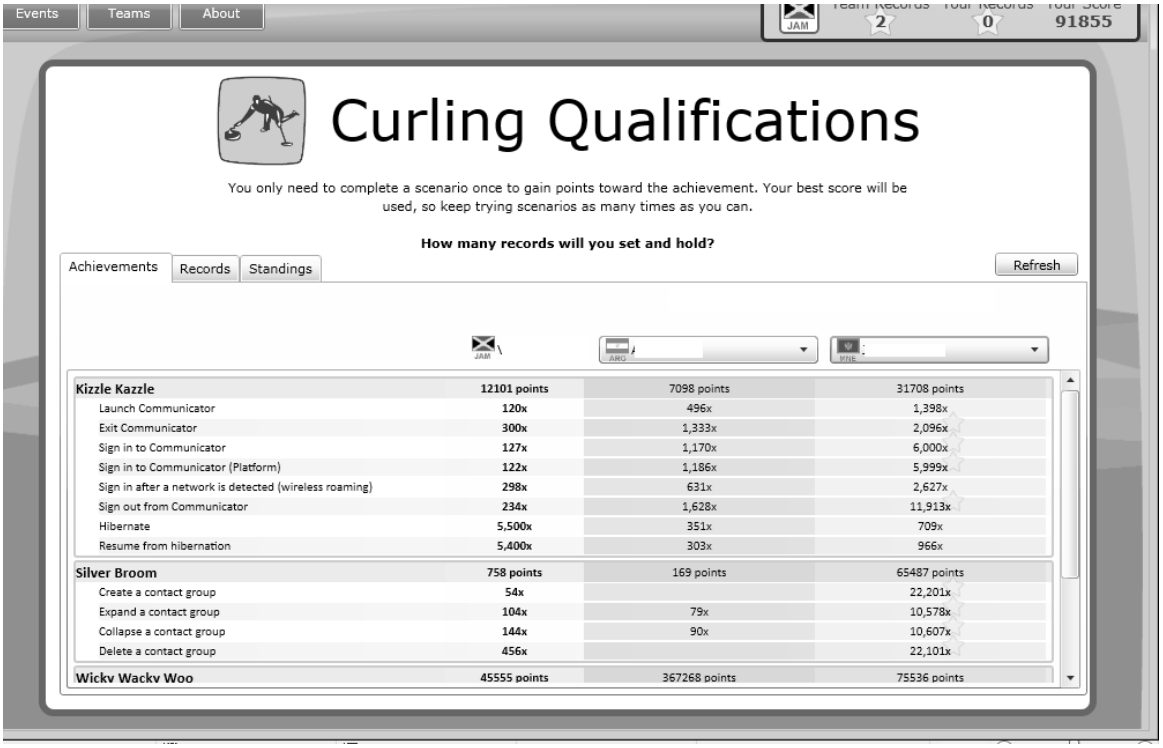


Figure 4- Screenshot of the achievements tab in the leaderboard

We also made it possible for a user to see the leaderboard for a specific scenario within the “Curling Qualifications” event. Figure 5 shows the leaderboard for the “Launch Communicator” scenario. This allowed participants to easily see what the highest score was and how they were doing on each scenario compared to their peers.

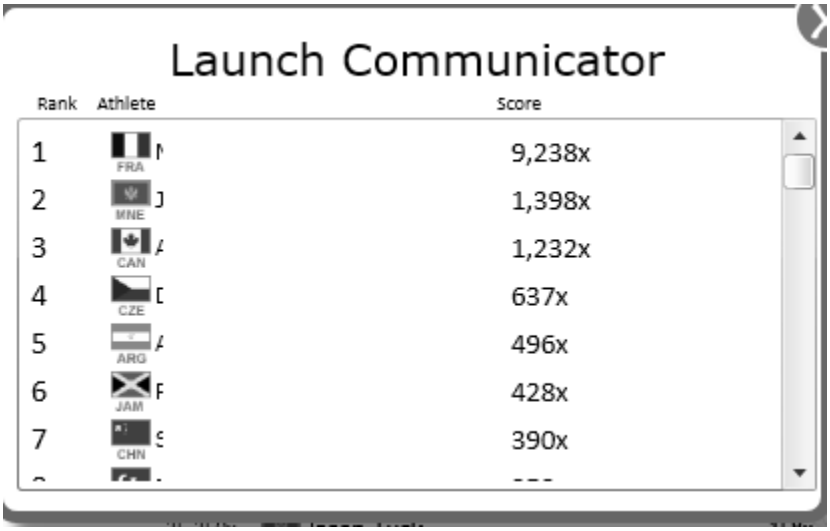


Figure 5- Standings for the "Launch Communicator" scenario

4.3. The “Spell Communicator” game

The “Spell Communicator Game” was the next phase of our experiment. It was advertised to dogfood users with instructions stating: “*Each of the letters in ‘Communicator’ is associated with a key scenario for Office Communicator and will light up when you have successfully completed it. After spelling Communicator, completing a bonus scenario will cause the Communicator “fishhook’ logo to light up, and fame and fortune will result. OK, so maybe not fortune, but certainly fame will be yours.”*

Once we had all the games in place and Application Monitor was known on the broader team, we encouraged developers and testers to add instrumentation for their features, so they could feature as letters in the Spell Communicator Game. This was a great way to improve the quality and quantity of instrumentation in Lync source code.

Figure 6 shows a couple of key UI elements of the game. Letters in blue (the letter ‘o’ in the figure) indicate scenarios that have been completed while hovering the mouse on a light gray letter causes it to darken (the letter ‘n’ in the figure) and display a hint on how that achievement can be unlocked. In a sense, this UI allowed a user to easily see his “score” and figure out how what other steps were needed to complete the game.

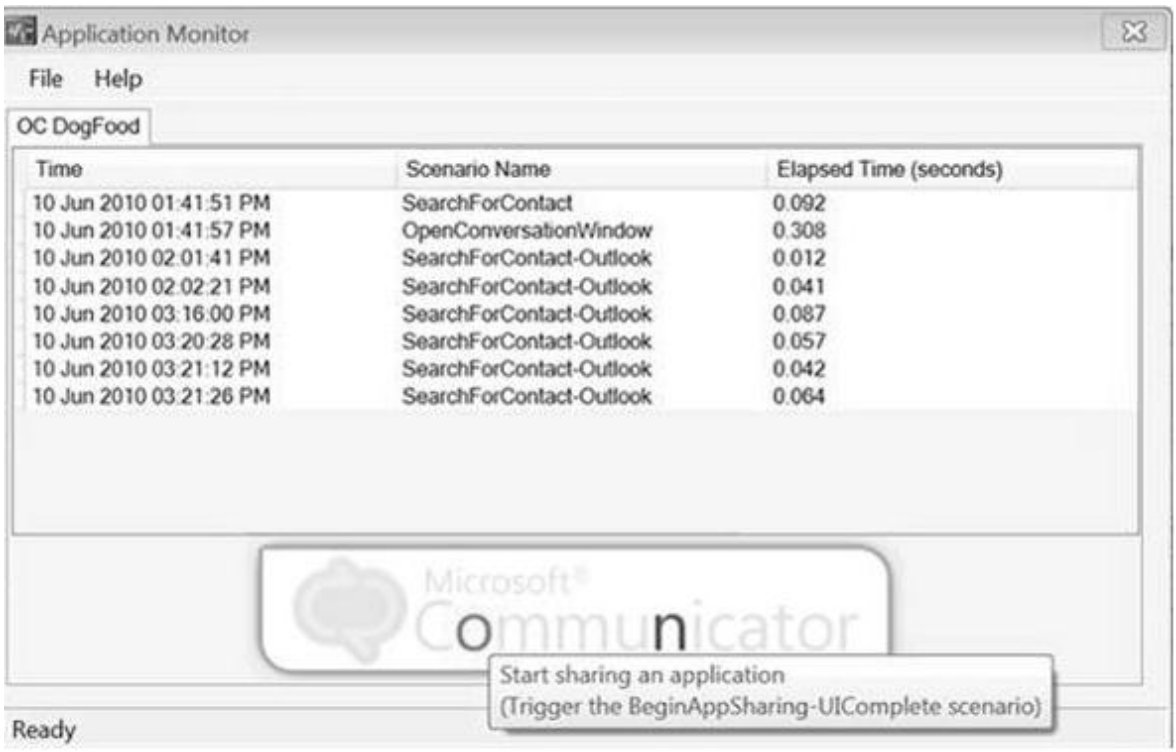


Figure 6- Screenshot of Application Monitor with the Spell Communicator game

The way Application Monitor was designed, it was fairly simple to have the scenario completion information sent to a WCF service which stored and tracked scenario completion information for each participating user in a database. The rules on whether or not an achievement should be unlocked were stored on the server and the server’s response based on these rules would indicate whether or not to light up an achievement. The achievement definitions and images were defined at the server as well and Application Monitor simply updated the appropriate image on the user’s machine. In a sense, this feature introduced a player vs.self game element into Application Monitor. Since these achievements were only unlocked per build, the user would have to start over again every time they upgraded their build. This was

a useful feature because it helped us get this basic level of coverage on each new build that users installed, essentially helping us smoke-test each build at low cost to the test team. In essence, we were getting the users to run a very specific set of tests for each build we released while providing them with some interesting and fun experience in return.

Making the achievements view more dynamic would only involve changes on that service to reply a different set of achievements data for different people, and making this configurable for each team/person would be implemented by creating a front end to change the DB data. Application Monitor code did not have to be modified to support a new set of achievements images or data.

5. Conclusion

Based on the amount of data we collected it was clear that we were successful in a) providing strong motivation for a certain section of dogfood users to increase their participation levels and b) in attracting a new set of dogfood users to try out Microsoft Lync. At its peak, we had about 200 users actively participating in the games and we got coverage on approximately 80 instrumented scenarios. We got a lot of useful information about which scenarios that were being executed most frequently and which were being executed the least frequently. The great thing about publishing this data was that testers and developers of Lync features that previously had no instrumentation for their scenarios were motivated to add instrumentation for their features as well. This helped increase coverage in terms of number of scenarios that could be tracked. In some cases, we also saw that the really competitive players even attempted to game the system by automating game scenarios so they could quickly get to the top of the leaderboard. While this behavior was not really conducive to the spirit of the game, it was still a fantastic outcome for us because this ensured very good coverage for those scenarios for almost zero additional cost to the test team.

There are also some issues we found along the way. The biggest issue by far was that the original instrumentation in Lync was added to help identify scenarios that were run in a deterministic fashion and in a controlled set of circumstances in the performance test lab. However, real word usage by dogfood users is very different and this exposed some holes in our product instrumentation for scenarios that did not terminate normally. For example, if a dogfood user cancelled the SignIn process or SignIn failed for the dogfood user, this was not detected correctly by Application Monitor and sometimes resulted in false positives i.e. Application Monitor would report completion of a scenario that the user really had not completed. Needless to say, this is a big problem as it strikes at the very heart of the system. The key takeaway from this is that the instrumentation needs to be added very carefully so as to capture all entry/exit paths for a given scenario. In addition, the logic in Application Monitor needs to be more robust to unexpected event ordering. The ideal implementation would use a carefully designed finite state machine to drive this logic, so as to always make the correct inference about which scenario was executed.

As mentioned earlier, another issue was that the need for users to perform a separate install posed a biggest hurdle to getting a more widespread adoption of the tool. If we were to do this again from the very beginning, we would definitely explore options to increase the incentives for dogfood users to install this tool by providing other useful functionality (for example, allow an easy way to report their feedback on Lync, allow an easy way for users to find out about known issues on the build of Lync they are currently running etc.) in the tool making it more attractive for users to consider installing and running it on their machine.

Acknowledgements

This project would not have been possible without key contributions and support from the following people:

- Mike Jackson – Software development engineer on the Lync team instrumental in the overall initial design of Application Monitor.
- Ross Smith – Director on the Lync client test and key supporter and evangelizer of productivity games. On his blog devoted to productivity games, he defines them as “a sub-category of serious games designed to improve the productivity and morale of people doing "regular work". (2)
- Joshua Williams – Senior tester on the Lync team and game master[8] for the various games documented in this paper.

References

1. Crowdsourcing. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Crowdsourcing.

2. *SCORE ONE FOR QUALITY! USING GAMES TO IMPROVE PRODUCT QUALITY.* **Williams, Joshua, et al., et al.** Portland : Pacific Northwest Software Quality Conference, 2009.

3. Eating your own dogfood. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Dogfooding.

4. **Smith, Ross.** *Productivity Games - Ross Smith .* [Online] http://productivitygames.blogspot.com/.

[8] The person who acts as the referee/organizer for conducting a game is known as a game master.

No Test Levels Needed in Agile Software Development!

Leo van der Aalst[1]
Lector "Software Quality and Testing" at Fontys University of Applied Sciences
l.vanderaalst@fontys.nl

Abstract

Testing is not only a vital element of agile projects, it is an integral part of agile software development. In traditional software development environments test levels like system test, user acceptance test and production acceptance test are commonly executed. In an ASDE, all team members have to work together to get the work done. All disciplines have to work together and support each other when necessary. There are no designer, developer, user or test teams in an agile project. In such an environment it does not make much sense to talk about system test and/or acceptance test teams. Instead all team members have to accept the feature (or user story or use case, etc.) with their own acceptance criteria in mind. For instance, an end-user should test the suitability and user-friendliness of the feature. Operations should, for instance, test the performance, the manageability and continuity of the feature. And the designer should test the functionality of the feature. In short, in an ASDE test levels are replaced by combinations of acceptor and quality characteristics important to the acceptor per feature. This approach requires a different mind-set of the team members, a different product risk analysis approach and a different view on establishing the test strategy.

Biography

Leo van der Aalst *has almost 25 years of testing experience and developed amongst others services for the implementation of test organizations, agile testing, test outsourcing, software testing as a service (STaaS), risk based testing, calculation of the added value of testing and test-governance.*

Leo is lector "Software Quality and Testing" at Fontys University of Applied Sciences (Eindhoven, The Netherlands) and he is co-author of the TMap NEXT® for result-driven testing and TMap NEXT® Business Driven Test Management books. He is also a member of the Dutch Innovation Board Testing and of the Research and Development unit of Sogeti Netherlands[2]*.*

Besides all this, Leo is a much sought-after teacher of international test training, a regular speaker at national and international conferences, and he is the author of several articles.

Speaker at conferences: a.o. STARWEST and PNSQC (both USA), Test Congress, Iqnite and Test Expo (all UK), Quality Week Europe (Belgium), SQC/Iqnite and TAV (both Germany), Swiss Testing Day (Switzerland), ExpoQA and QA&TEST (both Spain)

Cecile Davis *is co-author of this paper. She is a test consultant and has been involved in several agile test projects. She is a certified agile tester, RUP-certified, co-author of TPI NEXT® and founder of the SIG on agile testing within Sogeti. She is involved in several (national) SIG's related to this subject.*

[1] Rick D. Anderson thank you for your input. You were a great help!
[2] TMap NEXT and TPI NEXT are registered trademarks of Sogeti Nederland B.V.

1. Introduction

Structured testing can be perfectly integrated in agile development. This paper is intended for everyone with an interest in testing in relation to the agile manifesto and, specifically in how to test without test levels in an agile software development environment (ASDE). This paper does not describe in detail how to test without levels together with a specific agile method like scrum.

1.1 Vision on testing in agile environments

In agile processes a number of aspects pose a significant challenge to the traditional view of professional testing: lack of detailed requirements, reduced multipurpose process documents (test strategy, plans and cases etc.), always delivering working software, nightly integration and builds, user involvement, short time boxed iterations, potential technical requirements for testers, change of culture (self managed teams), distributed or off-shore teams and the fast pace of delivery.

It is important to realize that agile methodologies arose from the software developer's side. Agile is not a prescriptive discipline, hence, specific processes like testing, configuration management, build and release processes are not discussed. However neutral the agile manifesto is towards the test discipline, the methods that embraced the agile philosophy never focused on how to integrate the testing discipline thoroughly in ASDEs, nor did they consider what the consequences for test professionals would be. Therefore many organisations struggle with the implementation of testing in an ASDE and since testing is not only a vital element of agile projects but also an integral part of agile software development, we have defined a *test vision* in addition to the agille manifesto:

1. Use the agile manifesto as a starting point
   A test vision should based on the four values of the agile manifesto together with the twelve principles. This means that each and every proposed test activity must be in line with the agile manifesto and its principles.
2. Integrate the test activities in the ASDE
   a. The test activities must be integrated in both the development process itself and in the teams. Agile teams test continuously and all team members must be prepared to perform test activities. Although a professional tester should be part of the team, this does not mean that all test activities must be carried out by the tester.
   b. Testing should move the project forward. This means a shift from quality gatekeeper solely to collaboration with all team members to provide feedback on an ongoing basis about how well the emerging product is meeting the business needs.
   c. Test tools are increasingly important and should be used to support and improve the performance of agile teams.
   d. Test activities/acceptance criteria must be part of the definition of done.
3. Find balance by making well-considered choices
   The right balanced choices will always be context sensitive, which means that different factors like type of organization, type of project, business goals, available resources and available technology are taken into account.
4. Reuse values of the traditional structured test approaches, e.g. TMap NEXT® [3]
   The traditional test approaches are still very valuable, but sometimes they must be adapted to the agile way of working.

In this paper I'll explain at a high level how to put the above stated test vision into practice and in detail how to put a test approach without test levels into practice (*bullet 2a*).

[3] Aalst, L. van der, Broekman, B., Koomen, T., Vroon, M. (2006), TMap NEXT®, for result-driven testing, 's-Hertogenbosch: Tutein Nolthenius Publishers, ISBN 90-72194-80-2

1.2 Shift to an agile software development environment

In sequential software development lifecycles, the emphasis traditionally has been on defining, reviewing and subsequently validating the initial business requirements in order to produce a full set of high-quality requirements. Further levels of testing, such as system and user acceptance testing, are then planned to achieve coverage of these requirements and their associated risks.

This approach, combined with a full lifecycle testing strategy, utilizing effective document/code evaluation and other levels of development testing, such as unit and unit integration testing, can achieve very high levels of software quality.

However, in reality, projects invariably fail due to a lack of end-user involvement, poor requirements definition, unrealistic schedules, lack of change management, lack of proper testing and inflexible processes.

This traditional approach means that there often is a disconnection between users and testers. As a result, changes to requirements that often surface during the design or coding phase may not be communicated to the test team. This results either in false defects, or in a test strategy being incorrectly aligned with the real product risks. To combat this, traditional projects exert a lot of effort in managing change because in long-term projects, change is inevitable.

The agile software development approach, based on iterative development in which requirements and solutions evolve in combination and change is embraced, would therefore appear to offer a potential solution to the problems in a traditional approach.

Incremental software development processes have been specifically developed to increase both speed and flexibility. The use of highly iterative, frequently repeated and incremental process steps and the focus on customer involvement and interaction theoretically supports early delivery of value to the customer.

Agile software development is not a method in itself. It is derived from a large number of iterative and incremental development methods. Methods such as:

- Rapid Application Development (RAD)
  - 80's, Barry Boehm, Scott Shultz and James Martin
- Scrum - The New New Product Development Game – Harvard Paper
  - 1986, Ikujiro Nonaka and Hirotaka Takeuchi
- Dynamic Systems Development Method (DSDM)
  - 1995, DSDM Consortium
- eXtreme Programming (XP)
  - 1996, Kent Beck, Ken Auer, Ward Cunningham, Martin Fowler and Ron Jeffries
- Feature Driven Development (FDD)
  - 1997, Jeff de Luca

In 2001, in Utah, 17 representatives[4] of the above mentioned and other similar methods, met to discuss the need for lighter alternatives to the traditional heavyweight methodologies. In about three days they drafted the agile manifesto, a statement of the principles that underpin agile software development.

[4] The 17 authors of the manifesto were: Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland and Dave Thomas.

1.3 The agile manifesto

Agile software development has many different ‘flavours’. However, the foundation of any of these development methods known today can be found in the manifesto for agile software development. The agile manifesto consists of four values and twelve principles.

1.3.1 Four values

At http://agilemanifesto.org the manifesto is stated as:

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions	*over*	*processes and tools*
Working software	*over*	*comprehensive documentation*
Customer collaboration	*over*	*contract negotiation*
Responding to change	*over*	*following a plan.*

While there is value in the items on the right, we value the items on the left more!”

Some ways wherein the first value expresses itself are the frequent face-to-face communication between individuals, the self-regulation of the multidisciplinary teams and the team’s responsibility. The second value is followed by working in short iterations with working deliverables at the end of each iteration, by prototyping and by efficiency in documentation. The third value is supported by the involvement of customers in the team, frequent feedback and demonstrations. The last value can only be true if all disciplines are involved early, and the process is implemented based on a strategy that considers changes being inevitable.

1.3.2 Twelve principles

The twelve principles behind the agile manifesto are:

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity, the art of maximizing the amount of work not done, is essential.
11. The best architectures, requirements and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

2. How to put the test vision into practice

In this chapter the vision on testing in an ASDE will be discussed in more detail (see section 1.1).
In addition, some tips and hints will be given on how to put the test vision into practice.

The four, high level, test vision values are:
1. Use the agile manifesto as a starting point.
2. Integrate the test activities in the ASDE.
3. Find balance by making well-considered choices.
4. Reuse values of the traditional structured test approaches.

The above mentioned values are described in the following sections.

2.1 Use the agile manifesto as a starting point

When implementing an agile method, a lot of organizations pick one or two appealing values or principles from the agile manifesto and start to implement these favourite values. However, wise and sensible it may sometimes be to start with small steps, implementing only part of the values of the manifesto will not work in the end. It often happens that organizations stop implementing halfway or just do not spend as much attention to the other values as to the first. The manifesto is not a checklist you can pick from. The way these values are implemented is open, which is why there are different agile methods, but all values need to be implemented. And if an agile method is implemented partly, it will also have consequences on how to deal with test activities. Moving to agile is similar to any process change/improvement. That is, everybody needs to be clear on why the change, and the steps to be taken to change the process, is necessary.

First of all, to solve the issue of a test process that is not based on the agile manifesto as a whole, it is necessary that all stakeholders involved are familiar with the manifesto and the agile method that will be implemented. This seems very simple: just give them a book to read. However, it is very important that people really understand the values and their consequences. The manifesto and its principles stand for a mindset; it is not a checklist.

Furthermore, it is most important that people have the same understanding.
If everybody has to learn the values on their own, for certain, different interpretations will arise. That will lead to miscommunication and misunderstanding. The best and most complete solution is therefore, to start by appointing an expert as coach, who can give training and support. When everybody is used to the new way of working, this coach will no longer be necessary.

2.2 Integrate the test activities in the agile software development environment

In this section four aspects with respect to the integration of testing into an ASDE are discussed:
- Integrate in development processes and teams.
- Testing moves the project forward.
- Test tools are necessary.
- Testing is part of done.

2.2.1 Integrate in development processes and teams

In traditional development environments testing is often inserted at a later stage, although most test methods advise an early test involvement. In agile environments testing is, by nature, incorporated from the beginning and throughout the whole process. So, organizations need to think on how to deal with traditional test levels like system test, user acceptance test, end-to-end testing, and production acceptance test in an ASDE.

In this section three aspects with regard to continuous testing are discussed:

- Integrate testing activities with development activities.
- Test levels in an ASDE.
- The testers' role changes.

Integrate testing activities with development activities

Testing in agile projects cannot be seen as a separate activity but needs to be integrated in the entire process. Testing is not only a vital element of agile projects, it is an integral part of agile software development.

The underlying thought of the agile philosophy is to deliver business value as early as possible and in the smallest workable piece of functionality. To prove that the developed software works as desired *and* to prove business value, the user story and product need to be tested. To make agile work well, testing cannot be seen as a separate activity but needs to be integrated in the entire process.

Testers can add value in different areas by acting as a spider in the agile web. Besides being involved in testing, they will be involved in formulating the definition of done, in planning, in unit testing, in gathering all information necessary to form the test basis, in risk management, in retrospection, in test automation, etc. Therefore, the role of the tester is an important one. This early and high degree of involvement of testing in the entire project has a lot of benefits. For instance, including testing activities/acceptance criteria in the definition of done ensures that the software is considered 'done' only when the testing is fully 'done'.

A high degree of involvement at an early stage is essential for testing in agile environments. All disciplines, testing included, must be involved as soon as possible in the process. This is an essential part of agility, required, for instance, by the value in the manifesto of *responding to change.* In agile environments, changing the requirements can be done more easily than in a non-agile environment. If the change is discussed and agreed upon within the team, which would also include the customer, then no one can really oppose to the change. Testing, especially, is of importance in an early stage of changing requirements, because of the value it can add to impact analysis and risk analysis. The sooner testing can respond to changes and take actions, the better.

Also, responding to change raises a need for test automation, efficient and effective documentation and more face-to-face communication. These differences require a different approach and different skills from testers (see also section "the testers' role changes").

Test levels in an ASDE

In traditional software development environments, test levels like system test, user acceptance test, end-to-end testing and production acceptance test are commonly used. However, in ASDEs, all team members have to work together to get the work done. In such an environment it is not relevant to talk about system test and acceptance test levels/teams with managers and budgets of their own.

If there is a need for some distinction, instead of test levels, quality characteristics can be used per user story. This way, certain test activities can be grouped together. Quality characteristics can also be used to SMART-en up acceptance criteria in the definition of done. All team members have to accept the feature with their own acceptance criteria in mind. For instance, an end-user should test the feature suitability and user-friendliness. Operations should test the feature manageability, performance and continuity for instance. And the designer should test the functionality. In short, in an ASDE test levels are replaced by combinations of acceptor and quality characteristics important to the acceptor per feature. See figure 1.

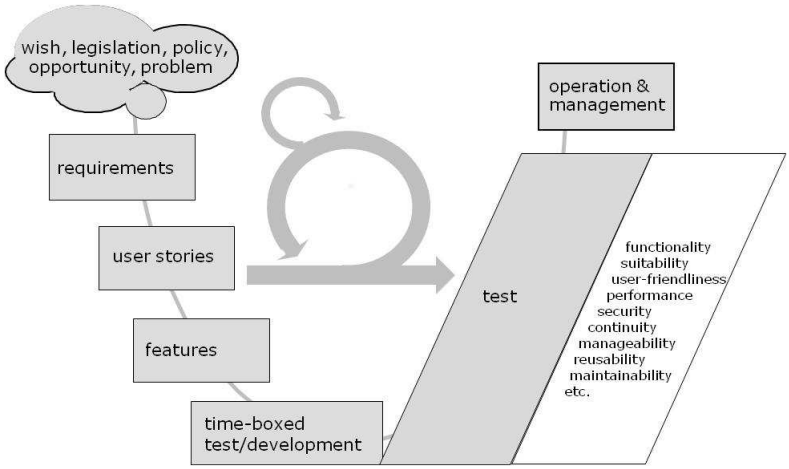


Figure 1: All test activities carried out by the agile team.

The characteristics for such an approach are:

- No seperate test levels, not in the iterations nor afterwards.
- All acceptors of the product must be present in the, relevant, iterations and prepare/execute their own tests (with or without help of the other project members)
- In the product risk analysis, the risk per combination of feature and quality characteristic should be determined by and per acceptor.
- After the last iteration the product is explicitely accepted by all acceptors.

However it is still possible to group certain test activities to a specific test level. One could say that all tests aimed at, for instance, testing the functionality of a feature is called the system test. And that all tests aimed at, for instance, testing the suitability and the user-friendliness of a feature is called the acceptance test.

Depending on for instance the availability of people, environments or for other reasons, it is also possible that certain test levels are not carried out by the agile project team itself, but it could be a test phase after the release is delivered or in parallel with the iterations. In this situation it's possible to make a distinction between iteration tests and release tests. See figure 2.

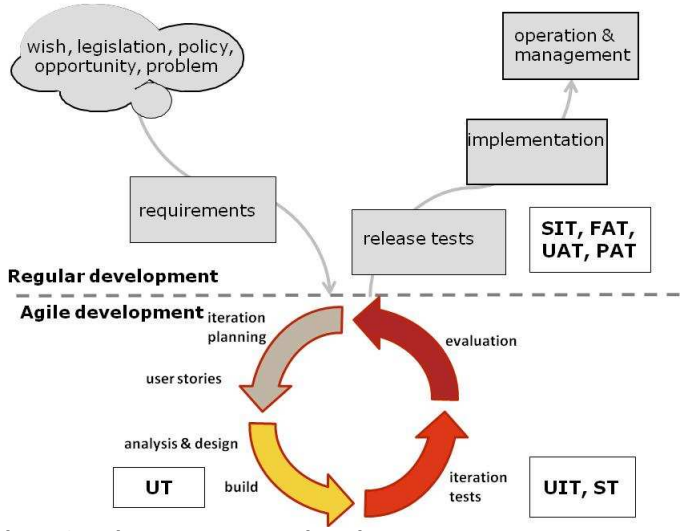


Figure 2: Release tests versus iteration tests.

And what about end-to-end testing? Is it possible to incorporate this test level in the agile process? This depends on the specific situation. Theoretically, end-to-end testing would benefit enormously from agile

software development. However, it will usually be difficult to realize the end-to-end test within the agile process. Therefore, it could be wise to implement end-to-end testing as a separate phase after the agile project is completed.

The testers' role changes
Agile project teams have to work together closely and the team should contain the people required to make the project successful. In agile methods, involvement of all disciplines, including testing, is part of the philosophy. Therefore, working with multi-disciplinary teams is very important. So a tester must be part of the agile team.

Working in multi-disciplinary teams also means that all team members must be prepared to fill in other roles if necessary. For instance, a developer must be willing to fill in a testing role if the need arises. Especially, of course, when there is time pressure on the test activities. On the other hand, testers must be willing to assist other team members in their activities where they can according to the projects needs. This can demand more technical skills and knowledge from a tester. Working together means two-way traffic.

The nature of the testers' role changes in iterative projects. Testers are no longer the high-profile victims, they are no longer the lonely advocates of quality. They are competent service providers, working in a team that wants to achieve high quality. The testers' role becomes richer and more influential. Agile methods require testers to be involved in the development project continuously and right from the start.

Testers need to have technical knowledge of the software they are testing and need
to understand the impact on automation as well as the functional implications. Some iterations may be development heavy, some automation heavy, some test heavy; the agile tester needs to be adding value in all instances.

The personal characteristics that are especially important for a tester working in an agile team are:
- Communicative
- A retrospective attitude
- Flexible
- Pro-active
- Creative but practical
- Thinking in solutions
- Customer-oriented
- Open-minded
- A team player
- Interfering with everything like a spider in a web

Professional testers should adapt to fit this different role and so provide additional value by not only focusing on finding defects but also fulfilling a team role.

2.2.2 Testing moves the project forward

On traditional projects, testing is usually treated as a quality gate, and the test team often serves as the quality gatekeeper. The result of this approach exist of long, drawn out bug scrub meetings in which different parties argue about the priority of the bugs found in test and whether or not they are sufficiently important to delay a release.

In agile teams, the product is built well from the beginning, using testing to provide feedback on an ongoing basis about how well the emerging product is meeting the business needs.

This sounds like a small shift, but it has profound implications. The adversarial relationship that some organizations foster between testers and developers must be replaced with a spirit of collaboration. This is a completely different mindset.

Testers must be pro-active and work with the business stakeholders to understand their real needs and concerns. In traditional environments, this is usually called ‘requirements elicitation’. In the context of agile development, the purpose of this discussion is not to gather a huge list of requirements but rather to understand what the business stakeholder needs from one particular user story.

During these discussions, the tester must ask questions designed to uncover assumptions, understand expectations around non-functional needs such as performance, reliability, security, etc., and explore the results the business stakeholder is requesting. This will improve the quality of the software product.

One, very effective, way of involving testers early is to introduce test driven development. This way, testers, programmers and business representatives can all learn from each other, while interpreting the requirements together.

2.2.3 Test tools are necessary

Over time, test tools have become increasingly important to the performance of agile teams. This is not just because teams sometimes have to be technically oriented, but because the right tools can help a team to become more efficient and agile.

If agile is about speed, efficiency and flexibility, then the role of automation in agile is to support this and remove mechanical, routinely and time consuming tasks. Due to the required speed of agile, management of test data and environments needs to be very efficient and effective with little if any room for unnecessary manual effort. Although this seems logical, it is still very hard for people working in an ASDE to decide upon the tools to be used and how these tools should be implemented and used. Tasks that can normally be automated within agile teams include:

- Build and integration process.
  Usually in agile teams, this process happens on a very regular basis (every night), resulting in a new build every day, with almost zero manual effort being put into this task. This requires good configuration management and build tools.
- Unit (integration) Test.
  These are part of the nightly build and integration process, allowing the development team to get instant feedback on the quality of their code. The execution of these unit tests requires no manual intervention.
- Static Analysis Tools.
  Instead of doing manual code reviews, the analysis tools review the
  code against coding standards to uncover defects. The manual reviews can be kept for particular types of defects or more complex code.
- Test data and environment management.
  Available tools can generate data to manage the test environment.
- Regression Testing.
  To be able to follow the quick pace of agile software development, agile teams cannot do without automated regression testing.
- Functional Testing.
  Until now, functional automation testing has focused on regression testing. However agile teams are pushing for functional testing to be automated earlier and earlier in the development lifecycle, so that it is the design of test cases rather than the execution that is important. The use of Model Based Testing (MBT) tools in combination with the automation of test case execution is a (almost) perfect solution for functional testing in agile environments. Test execution should be automated as much as possible.

2.2.4 Testing is part of done

In traditional software development environments with strict boundaries between development and test, it often happens that user stories are declared ‘done’ before it is tested properly. Of course, agile teams only count something as ‘done,’ when it has been implemented, tested and bugs have been fixed.

Incorporating testing in the definition of done gives more control on iteration planning, on testing risks, more early involvement for testers and therefore more grip on the test process.

2.3 Find balance by making well-considered choices

The most important consideration when using an agile software development method, is finding the right balance in the choices that have to be made. When you look at the four values of the agile manifesto, you could think about a balanced choice for each value between the left and right item.

In practice, however, we see agile project teams struggle mostly with finding balance between:
- Working software and comprehensive documentation.
- Covering the risk and the available time and money.

Therefore these two balances will be discussed in the following sections.

2.3.1 Find the balance between working software and comprehensive documentation

Does agile mean the end of all documentation? To those looking at adopting agile practices, especially if they come from more traditional project management backgrounds, agile can seem quite loose, especially in the area of documentation.

The agile manifesto values working software over comprehensive documentation. And in agile projects working software is perhaps the ultimate quantification of your projects status. This may take some time to get used to.

Agile methods are all in favor of documenting only what is necessary. They simply value working software over comprehensive documentation. However, what level of documentation is necessary may vary per project. Besides that, just deciding on what to document and what not, is not enough. Whenever decisions are made on, for instance, the extent of documenting requirements, it must be thought over how the information about these requirements that is not in the documentation is communicated between the different “requirements stakeholders”. There should be a balance between documentation and communication so that important information does not get lost.

In addition, there is a number of principles behind the manifesto that elaborate on this value:
- Working software is the primary measure of progress.
- The teams’ highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

Sometimes end-user documentation is used as one of the criteria for ‘done’. It can also be used to document decisions for team continuity. And documentation may even be needed for regulatory compliance.

With regard to the quantity of documentation to be produced, the team could be asked to answer two simple questions:
- Does the documentation add value?
- Is the team better off writing it?

2.3.2 Find the balance between risks versus time and money

An agile project, like any other project, never has unlimited time and money for testing purposes. Such constraints in terms of time and money represent constraints on the test result to be achieved and

therefore reduced coverage of the product risks. As such it is important to make well-considered choices in relation to the optimum division of the available time and money across the user stories that require testing. In fact we need to determine product risks and test strategy in an *agile business driven test management way*.

Based on the insight resulting from the product risk analysis, high risk user stories can be tested more thoroughly than those representing a lower risk. A product risk is the chance that the product (user story) will fail in relation to the expected damage if it does.

Product risk = Chance of failure * Damage

When performing the product risk analysis, all stakeholders should be member of the agile team and participate in the analysis.

In an agile environment it is difficult to perform a product risk analysis for the release as a whole. There is just not enough detail available of individual user stories. So, in practice a product risk analysis can be performed at the start of each iteration. Since an iteration will deliver a few user stories only, it does not take much time to perform the analysis (it could be done on a whiteboard).

The easiest way to proceed is the following:
1. Gather all team members.
2. List all user stories of the current iteration (on the whiteboard for example).
3. Ask all participants which quality characteristic(s) per user story is (are) important to the participant.
4. Determine per user story and quality characteristic what the “Chance of failure” and the “Damage” are. If figures are used. The risk is simply the product of the figures.

At this moment the product risk analysis is done and the whiteboard could look like this (see figure 3):

User Story	Characteristic	Stakeholder	Damage	Chance of Failure	Risk Class
US 1	Functionality	Person A	3	3	9
	User-friendliness	Person B	2	1	2
US 2	Functionality	Person A	2	2	4
	Security	Person C	3	2	6
US 3	Functionality	Person A	2	1	2
US 4	Performance	Person C	2	1	2
US 5	Performance	Person C	1	1	1
US 6	Functionality	Person A	2	2	4
	Suitability	Person B	2	2	4
..	..		..	..	..

Figure 3: Risk table.

The next step is to determine the testing strategy. In this activity, the outcome of the product risk analysis is used to determine which combinations of quality characteristic and user story are to (must) be tested with what test intensity. In order to make the table more concrete it is possible to extend the table with a column “Test Design Technique”. This will help the team members with testing roles, preferably the stakeholders themselves, in creating the test cases (see figure 4).

In this example all user story and quality characteristic combinatios are carried out by the agile project team members. However, as stated before, it also possible to group certain combinations to a test level which could be executed in parallel to the iteration or after the delivery of the release. For instance, all combinations with the quality characteristic “functionality” in it, could be grouped to a system test level.

User Story	Characteristic	Stakeholder	Damage	Chance of Failure	Risk Class	Intensity	Test Design Technique
US 1	Functionality	Person A	3	3	9	●●●	ECT-MCC
	User-friendliness	Person B	2	1	2	●	SYN
US 2	Functionality	Person A	2	2	4	●●	ECT-MCDC
	Security	Person C	3	2	6	●●	SEM-MCDC
US 3	Functionality	Person A	2	1	2	●	DCoT-EQ
US 4	Performance	Person C	2	1	2	●	EG
US 5	Performance	Person C	1	1	1	●	EG
US 6	Functionality	Person A	2	2	4	●●	ECT-MCDC
	Suitability	Person B	2	2	4	●●	PCT-TDL2
..	..		..	..	..		

Figure 4: Test strategy table[5]**.**

When using these tables one can add other columns to it. Like “Test Cases created” (Y/N), “Test Cases executed” (Y/N) and “Tests Passed” (Y/N). And everone has only to look at the whiteboard to see the (test) status (see figure 5).

User Story	Characteristic	Stakeholder	Damage	Chance of Failure	Risk Class	Intensity	Test Design Technique	Test Cases created (Y/N)	Test Cases executed (Y/N)	Tests Passed (Y/N)
US 1	Functionality	Person A	3	3	9	●●●	ECT-MCC			
	User-friendliness	Person B	2	1	2	●	SYN			
US 2	Functionality	Person A	2	2	4	●●	ECT-MCDC			
	Security	Person C	3	2	6	●●	SEM-MCDC			
US 3	Functionality	Person A	2	1	2	●	DCoT-EQ			
US 4	Performance	Person C	2	1	2	●	EG			
US 5	Performance	Person C	1	1	1	●	EG			
US 6	Functionality	Person A	2	2	4	●●	ECT-MCDC			
	Suitability	Person B	2	2	4	●●	PCT-TDL2			
..	..		..	..	..					

Figure 5: Test progress table.

The test strategy is the foundation for every test action, activity, process or project. It holds the justification of what will be tested and how this will be done. Proper risk analysis drives this justification: “No Risk, No Test”. The test strategy describes the test goals, how they will be approached and how (product) risks are covered. Risks influence priorities in agile methodologies. Areas that carry the greatest risks need to be prioritized highly. Furthermore, risks will trigger the noted flexibility of agile environments in deciding what to do and what not to do in terms of time, costs and quality. The test strategy thus facilitates the whole agile team.

[5] ECT-MCC: Elementary Comparison Test-Multiple Condition Coverage, SYN: Syntactic Test, ECT-MCDC: Elementary Comparison Test-Modified Condition/Decision Coverage, SEM-MCDC: Semantic Test-Modified Condition/Decision Coverage, DCoT-EQ: Data Combination Test-Equivalence Classes, EG: Error Guessing, PCT-TDL2: Process Cycle Test-Test Depth Level 2 Want to know more? Take a glimpse at TMap NEXT (ISBN 90-72194-80-2).

2.4 Reuse and adapt the values of structured test approaches

Many people think that agile projects are chaotic, unorganized and uncontrolled. On the contrary. Agile projects will not be successful when they lack discipline or structure. Because of this prejudice, however, structure is often thrown out as is the baby with the bathwater.

A structured testing approach offers the following advantages:

- It delivers insight into, and advice on, any risks in respect of the quality of the tested system.
- It finds defects at an early stage.
- It prevents defects.
- The testing is on the critical path of the total development as briefly as possible, so that the total lead time of the development is shortened.
- The test products (e.g. test cases) are reusable.
- The test process is comprehensible and manageable.

When implementing an agile method, organizations often find it difficult to decide on what practices to keep and what practices to throw out. Usually, organizations find it easier to start afresh without much consideration for the consequences of throwing out old structures, and very often the baby is thrown out with the bathwater.

When organizations refrain from this decision process, they might save time at the beginning, but in the end the result can be loss of insight into the quality of the tested system, lacking reusability, lacking maintainability, more defects and delays in testing due to an unmanageable test process and most of all loss of all the advantages of an efficient running agile project.

To be able to decide upon which test structures and activities to keep and which to discard, it is necessary to have a good knowledge about the agile method to be implemented. An expert as coach, who can give training and support and who is aware of all the issues coming up during a transition from one method to another is almost indispensable. Usually, many practices and procedures, whether or not adjusted, can be reused.

3. Conclusion

Testing in agile software development environments ensues a different way of working. Moreover, a different mind-set is necessary. It will not be sufficient to follow a list of rules, applicable in an ASDE. A professional agile tester will need a thorough understanding of the agile approach along with a readiness to step out of the test compartment. Doing whatever is necessary in an effective and efficient way, in order to be a valuable agile team member, must be the goal of an agile tester. In general a structured test approach can support the agile tester to achieve this. In this paper the test approach without test levels is discussed in more detail. The characteristics for this approach are:

- No separate test levels, not in the iterations nor afterwards.
- All acceptors must be present in the, relevant, iterations and prepare/execute their own tests (with or without help of the other project members).
- In the product risk analysis the risk per combination of feature and quality characteristic should be determined by and per acceptor.
- The test strategy (= test intensity/test design techniques) is also determined per combination of feature and quality characteristic.
- No more (many pages) test plans needed, but just one table on the whiteboard.
- After the last iteration the product is explicitly accepted by all acceptors.

Application Compatibility Framework - Building Software Synergy

Ashish Khandelwal (Ashish_Khandelwal@McAfee.com)
Shishira Rao(Shishira_Rao@McAfee.com)
Amrita Desai(Amrita_Desai@McAfee.com)

Abstract

Building a healthy inter-product alliance in a software ecosystem requires a great deal of effort. Our paper focuses on simplifying a product compatibility testing and helps a tester to find compatibility defects early in the testing life cycle. It demonstrates a "Framework driven approach" for improving Product Quality from compatibility standpoint which is proven, tested and sustained over releases of a product.

On the basis of underlying testing methodology, we have divided our compatibility approach in three testing types.

a) OS compatibility Testing
b) Third Party Compatibility Testing
c) Endpoint Compatibility Testing

As a standard practice, we follow compatibility model described in our paper and make sure that our product passes the product compatibility testing procedure. With these three testing types in place, not only we have identified risky areas early in the test cycle but also succeeded in influencing product management decisions based on our results and analysis. Here are few of the strategic points which a compatibility tester can leverage upon from this paper.

a) Align to a standard compatibility model
b) Systematic understanding of three compatibility testing types
c) Overview of sample results, case studies and analysis

Biography

Ashish Khandelwal *has more than 6.5 years of Software Testing experience. He holds a B.Tech degree from IIT Kanpur and works as a Senior QA Engineer with the McAfee Host DLP product solution group. He has contributed to multiple international conferences and published papers on Compatibility & Security Testing.*

Shishira Rao *has more than 7 years of Software Testing experience. She holds a B.E. degree from VTU and works as a Senior QA Engineer with the McAfee Host DLP product solution group. Her testing and QA experience includes a focus on Process improvement, Compatibility testing and Strategic planning.*

Amrita Desai *is a QA Engineer at McAfee and works with the Host DLP Product solution group with more than 3 years of experience. She has a Bachelor's Degree in Computer Science from RGPV University. Her testing and QA experience includes a focus on Black box testing, Compatibility testing and Soak testing.*

1. Introduction

How many times have you encountered that your customer has escalated issues in their environment after shipping your product? Have you made an analysis of the number of sustaining patches and hotfixes that have been released in order to maintain the synergy between different applications in the customer environment?

Compatibility testing is the process to determine the impact of conflict between multiple applications in a computing environment and to maintain information system functionality as intended. It is a part of software non-functional testing , which is conducted on the application to evaluate the application's compatibility.

So, how does compatibility testing relate to synergy?

In a technical context, synergy in an environment is established when different applications working together produce results, which cannot be obtained by an application alone. For instance, Open Office is an application suite that can be considered as a fair replacement of Microsoft Office in low budget projects. But do you think the alternative would have worked if Open Office does not support compatibility with Microsoft Office files? Definitely not!!

The ability of these two office suites to work together in harmony makes up this synergized environment.

This paper talks about developing a software ecosystem where applications not just work without conflicts but also work for each other. Here, we are trying to explore the use of model-driven testing techniques and tools to increase quality and make the testing process more systematic and efficient. This presentation will describe the introduction of such technology.

2. Application compatibility – Need of the hour

Historically, in case of functionality testing, corners get cut and often the last phase “Stabilization” suffers the impact of lack of compatibility testing. Customers are dissatisfied, teams are exhausted and quality is compromised in the drive to meet the promised date. To overcome these issues, compatibility testing plays a vital role in minimizing the consequences of lack of stabilization.

“Every Product is subject to Risk”. Let us consider a couple of real life scenarios where compatibility testing could have played a major role in identifying the risks much before it has turned into calamity.

XP Alternative to Vista PCs

While Microsoft is still pushing Vista hard, the company is quietly allowing PC makers to offer a "downgrade" option to buyers that get machines with the new operating system but want to switch to Windows XP. One of the biggest challenges, for both consumers and businesses are Vista’s incompatibility with other applications, intermittent crashes, memory needs and hefty graphics.

Notably the effects could have been alleviated by employing measures that would have detected most of the compatibility issues with other Microsoft applications and thus taken care of memory needs.

Website render wrong in some browsers

Even with all the recent strides in more standardized browsers, we've learned that you still can't let your guard down when it comes to testing cross browser compatibility

Few web based products maintain their independent spirit. Lack of consideration to cross browser compatibility testing brings a bad impression to the users and thereby hampers the growth of the company.

Who will do compatibility testing?

Certainly, there are no pre-requisites for a tester who wants to perform compatibility testing. The right amount of attitude with zeal to learn could be the starting point for a compatibility tester.

We would like to introduce you to our product environment. We are a team of 8 QA Engineers and 12 Developers distributed across multiple location. The skill-sets and experience levels vary across the team, ranging from Software Engineers working in Quality to QA Analysts with deep domain knowledge. As an initiative, we (a team of 3 functional testers) took up the challenge to find compatibility flaws in our product. The next section describes the foundation of ***A****pplication* ***C****ompatibility* ***F****ramework.*

Let us now start with defining and describing our Application Compatibility Framework.

3. Application Compatibility Framework

This framework highlights the way we perform product compatibility testing and how it fits into the product life cycle. It includes three types of compatibility testing model, OS, 3rd Party & Endpoint, and depicts the correlation between them.

The selection of type of compatibility model in a release cycle is based on various factors such as new OS version support, major or minor release of the product, affected modules of the product and endpoint release dates.

OS compatibility model comes into picture when product management announces the support of an Operating System or a service pack of a pre-supported Operating system is released. In few cases, when application's new modules are affected by OS changes, we perform a regression cycle of OS compatibility suite.

We ought to consider 3rd Party Compatibility testing in case of major releases of the product. However, in some cases with small releases we take prioritized list of applications to be tested.

Any host-based product managed by a centralized server is defined as an Endpoint. Endpoints release date is a key factor in defining plan for Endpoint Compatibility model. We identify the release dates of endpoints which are in the time frame of our product release cycle and strategize to perform endpoint compatibility testing.

Below figure shows the relation and overview of the compatibility framework.

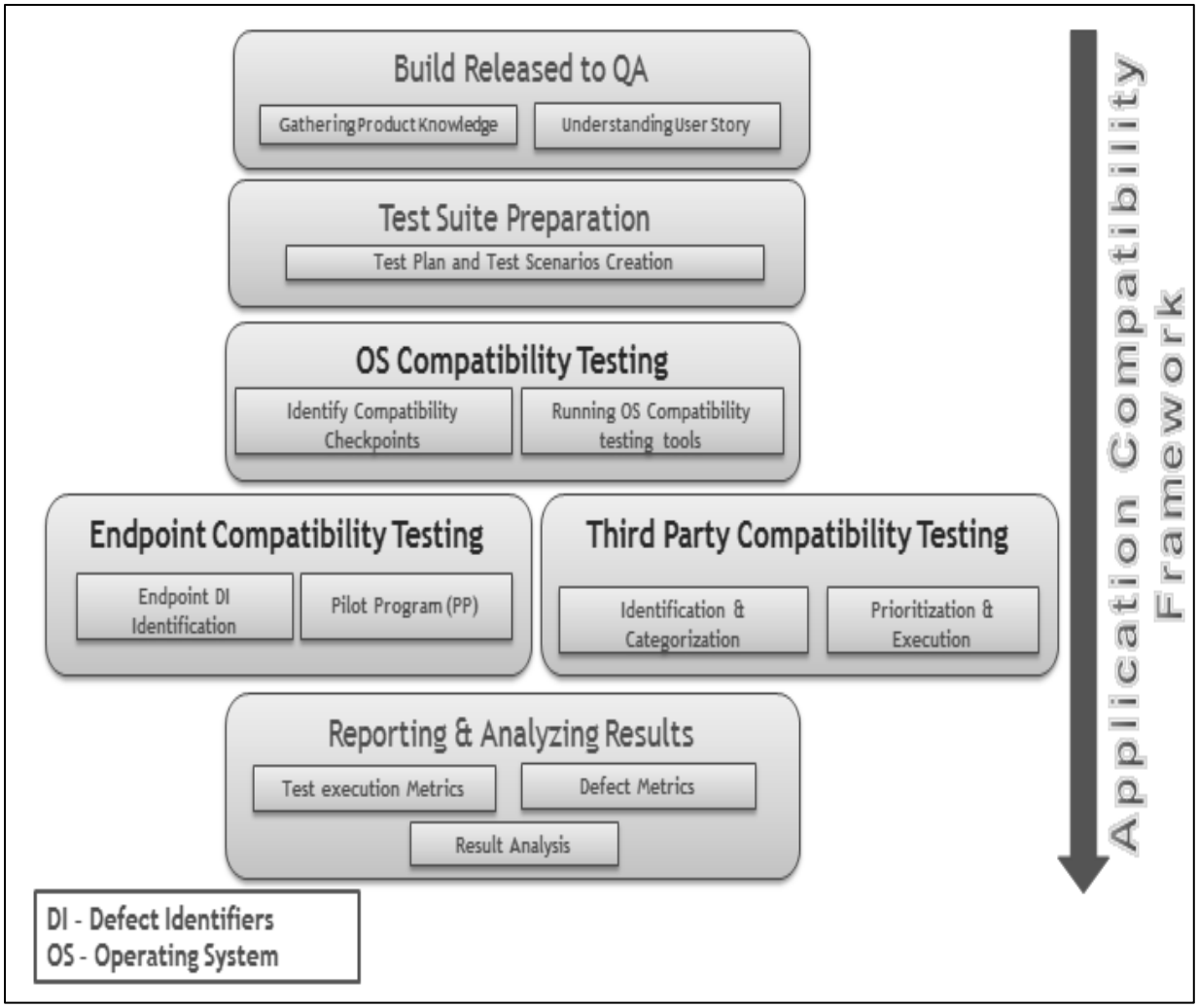


Figure 1 Application Compatibility Framework

In every major release, we plan to ensure that there is a decrease of at least 25% of defects found on customer site concerning compatibility and at least 10% growth in number of compatibility defects logged via compatibility framework. We also aim to build our 3rd Party application portfolio such that no defect occurs outside those applications.

Next three sections will talk in detail about the three compatibility-testing models and the individual role they play in defining the framework.

4. Operating System Compatibility Testing Model

Operating System (OS) design change can propagate numerous compatibility defects in a product. The purpose of this testing type is to uncover such issues that occur due to these enhancements.

For instance, from Windows XP to Windows Vista, Microsoft designers have performed major changes in the system architecture. Although these changes were incorporated to improve the performance and design, they also lead to multiple compatibility issues due to inability of other products to cope with the changing environments.

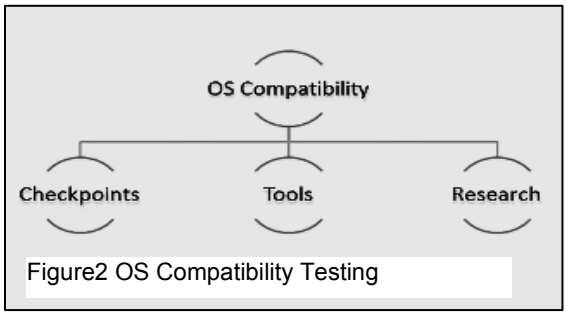


Figure2 OS Compatibility Testing

As a result, with frequent kernel upgrades and service packs release, it has become essential to strategize the compatibility testing approach that uncovers these issues early in a product cycle.

So, how should we perform it? What are the various parameters that will help uncover the issues related to changes in OS?

Figure2 highlights the various characteristics of OS Compatibility testing as explained below:

Checkpoints

These are the various features in OS which can be point of conflict with the application under test. For e.g., User Account Control (UAC), AppLocker, BitLocker.

Research Portability

Research done on Operating System and Service Packs is independent of the knowledge of the product. Later, compatibility checkpoints are identified based upon the application behavior.

For instance, Product A extends the support for Windows7. Thereby, research data gathered by a test engineer for Windows7 will contain a common study that can be shared across other teams, who can utilize them to gather their product specific checkpoints in their extended release for Windows7.

Hence, we can say that identification and study of checkpoints is portable across all products, provided they share a common support for an OS.

Tools

OS compatibility tools play a major role in identifying potential conflicts. OS vendors release them on purpose, as they anticipate compatibility issues. Few of the tools are provided by Microsoft like Driver Verifier, Application Verifier and IE Compatibility test tool.

Let us understand OS compatibility model with the help of a case study. Below you can see four stages of the model.

4-Step OS Compatibility Testing Process

Identify

In this stage we identify the OS or Service Pack for which our product enhances the support.
Let us say, Application XYZ, which contains add-ons for Internet Explorer & Microsoft Office and drivers for device blocking and file filter, starts support for Windows Vista.

Research

In this stage, we research about the compatibility checkpoints using various resources such as

a) OS Release document for new features
b) OS known compatibility issues published by the vendor
c) ACT Tool community/vendor assessment (only applicable to windows OS)
d) Online OS Release forums and newsgroups

We gather details of all compatibility checkpoints that may conflict with the application under test. For instance, in our case study analyzing checkpoints for Windows Vista leads to the following list:

a) Windows Vista UAC
b) BitLocker
c) Internet Explorer Protected Mode
d) Windows Resource Protection
e) Fast User Switching
f) User Interface Privilege Isolation

Later, we design the scenarios to test based upon these compatibility checkpoints.

Perform

All scenarios that are designed in previous stage are executed here. Here, we also use help of compatibility tools to uncover compatibility issues.

Coming back to our case study, Application Verifier can be used against XYZ to uncover few OS compatibility risks based upon different checks such as I/O Verification, Low resource simulation and deadlock detection. Likewise, we can use IE Compatibility test tool to verify if the changes in OS Architecture did not hamper any functionality of application's add-on for IE.

Report

Lastly, we log all the issues encountered and record the results for further analysis.

5. Third-party Compatibility Testing Model

"Third-party software is software which can be used in conjunction with some software or hardware but is not associated with either the manufacturer or the user. The testing of Third-party software is Third-party compatibility testing"

Let us take a quick example that explains 3rd party compatibility issues. Add-ons that get integrated in Internet Explorer (IE) are specially designed to perform extended functional tasks. Likewise, IE provides a common platform where these add-ons sit and can be seamlessly integrated. However, IE and add-ons will have a common code base that could result in potential compatibility issues.

Most of the 3rd Party Application incompatibilities arise due to:

1. Widely use of Internet for activities such as browsing, social networking and so on.
2. Changes in the implementation model of 3rd Party applications
3. Adoption of newer industry standards which invalidates older mechanism

With 3rd party compatibility model, we stay on top of these issues and try to mitigate the long-term effects they may cause.

Application Identification

Millions of applications exist but is it necessary to test with all of them? Do you know which application could affect your product and vice versa? Does your inventory include the applications installed on your customer environment?

Application Identification answers your questions where we prepare list of applications, which we see can cause a potential conflict with our product. The data is gathered from various sources including:

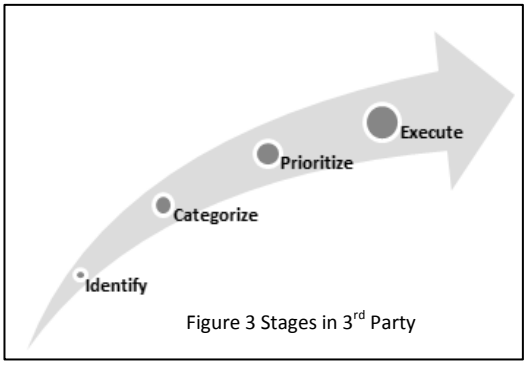


Figure 3 Stages in 3rd Party

1. Application Compatibility Toolkit (ACT) to identify applications installed on a specific system.
2. Tools to identify application installed in customer environment
3. Internally reviewed and installed product repositories
4. Any Enterprise Inventory tools

In practice, with the above criteria, we identified close to 200 3rd party applications. Noticeably, we discovered that 90% of the compatibility issues identified till now in our product involve these 3rd Party applications.

Application Categorization

Now that we have an inventory list from Application Identification, how do we test with these applications? We definitely need some guidelines and a common criterion based upon their functionality and features.

Generally, the applications will fall into one or the other category due to its functional behavior.
Using Application Categorization, we group applications into different categories based upon their underlying technology. For instance, IE, Firefox & Chrome will fall into "***Browser***" category and WinRAR, WinZip & 7zip will fall into "***Archiver***" category.

Once we have categorized the identified applications, next steps include:

- Creation of guidelines per category to create a test suite. This includes identifying the conflicting areas between applications.

- Preparation of Compatibility plan and creation of scope based upon release cycle. This includes the high-level plan keeping in consideration resource and time availability.

Application Prioritization

Is it practical to test all the applications that you categorized in a release cycle? Will you have that bandwidth to test with all pre-selected products from last phase?

We follow here the basic principle of prioritization to optimize the output with minimum effort. In Application Prioritization phase, we prioritize applications based on three factors that streamline efforts to cover most critical applications early in testing cycle. They include

a) Popularity Hits
b) Defect Fraction
c) Customer Escalation Fraction

Let us see below each in detail with the help of a case study on Browsers Category. To simplify, we have identified only three most popular browsers viz. Firefox, Chrome and IE.

Popularity Hits (PH)

This section is to measure how popular the application is with respect to other applications in the same category.

We derive this by dividing the number of hits of an application by total no of hits of all the applications in that category. Hits of any application are the number of results of that particular application on a popular search engine. So, in our example let's say

Firefox has 801m, Chrome has 622m and IE has 1.82b hits. Now Priority Links of Firefox can be calculated as

#Hits of Firefox/Total #Hits = [801]/ [801+622+1820] = 0.246

Defect Fraction (DF)

Here, we give priority points to an application based upon its historical defect data. If an application has more defects compared to other applications in that category, priority points of that particular application will be more. It is calculated by

#of Defects of an Application/Total # of Defects of All Applications under that category

E.g. Firefox has 4, Chrome has 3, and IE has 3 defects

Defect Fraction of Firefox = [4]/ [4+3+3] =0.4

Customer Escalation Fraction (CF)

This factor gives priority points to an application based upon the escalations observed from the customer environment. It is calculated by

Of Escalations of an Application from Customer / Total # of Escalations of All Applications under that category

For E.g., Firefox has 1, Chrome has 0 and IE has 1 escalation.

CF of Firefox = [1]/ [1+1] = 0.5

Priority Points

Once we identify all three factors of an application, we calculate Priority Points in the following manner.

$$PP = \left(\frac{PH}{TPH}\right) + DF + CF$$

PP – Priority Point
PH – Popularity Hits (Of Application)
TPH – Total Popularity Hits
DF – Defect Fraction
CF – Customer Escalation Fraction

Figure 4 Priority Points calculation

So, in our example:

Firefox Priority Points = 0.246+0.4+0.5 = 1.146
Similarly, IE & Chrome Priority Points can be calculated as 1.361 and 0.492 respectively.

The higher the Priority Points of an application, more critical the application for us to test. So, in Browser category we have IE, and then followed by Firefox and Chrome in prioritized application list.

Based upon the factors such as availability of resources and time, major or minor release cycle, we can take a call to choose high, medium or low priority 3rd party applications in a category to test.

Execution and Reporting:

This is the final phase of 3rd Party compatibility model. After going through identification, categorization and prioritization, finally we come to this phase where as per the scope we execute the test suite of prioritized products for all categories. During execution we match the actual and expected results and raise an alarm by reporting the issue in case of any deviation.

Let us understand this model with the help of a case study. We have identified two categories Browsers & Antiviruses and applied the different stages of the 3rd party model.

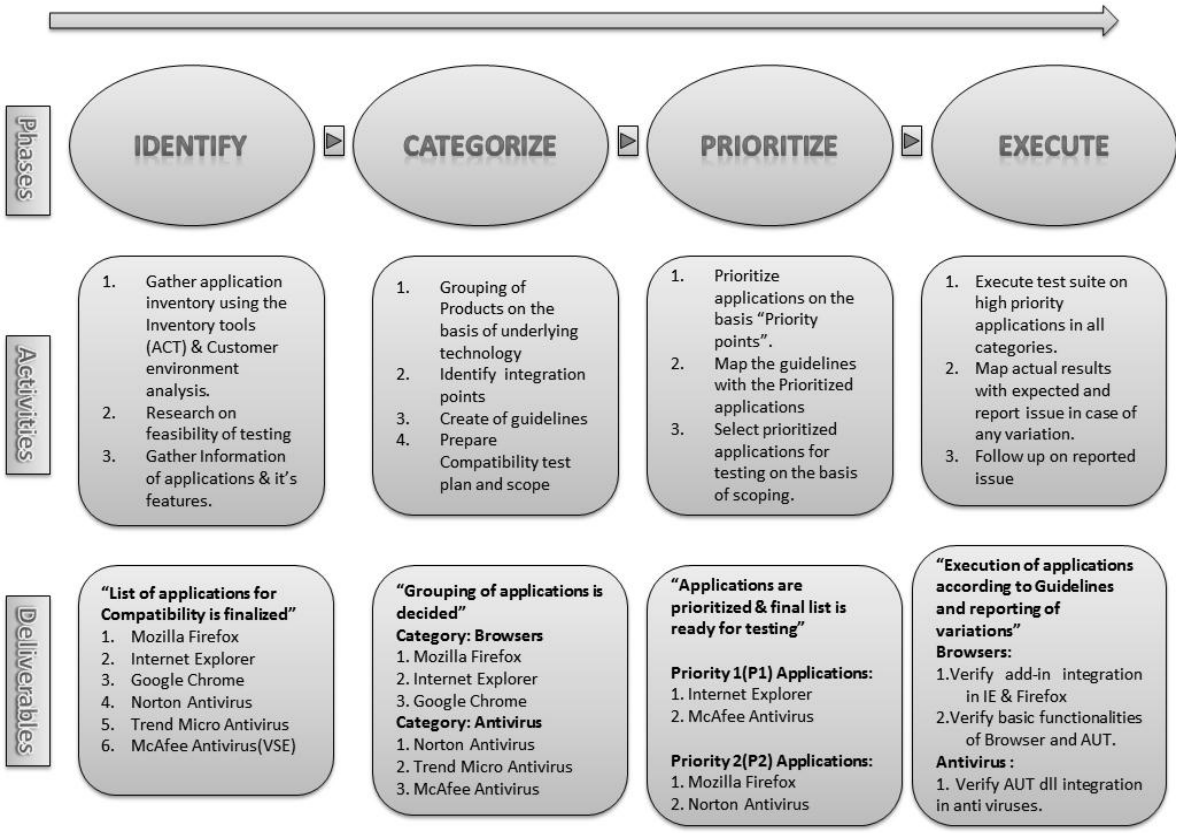


Figure 5 3rd Party Model Case Study for Browsers & Antiviruses

6. Endpoint compatibility testing model

"Endpoint compatibility testing is a process that enables synchronization of endpoint products to run on a computing environment."

In a customer environment, it is important to ensure that the products designed by the same vendor are in harmony. To do this, we have designed endpoint compatibility model that is helpful in identifying the potential conflicts among these products.

To understand this model, one need to understand the following key attributes.

Inter-team collaboration

Products from the same vendor give more scope for doing compatibility testing and that can be leveraged to make your plan effective. We acquire knowledge by internal training, formal or informal discussion with the peer product teams and in-house technical forums that can be used to develop key feature documents (Feature Flash or Knowledge Base Articles) and help us to design compatibility test suites.

Team-Paired Testing

Here, team members from different teams sit together and conduct an exploratory or ad-hoc session to test potential conflicting scenarios between their products.

Defect Identifiers:

Inputs from inter-team collaboration and team-paired testing help in identifying the key areas where the product's functionality ceases to co-exist. These key areas we call as "Defect Identifiers". For instance, a ***Reports*** module exists in a product whose codebase and libraries are shared across all endpoints in an organization. The UI interactions and code libraries of this ***Reports*** module can be considered as defect identifiers.

Pilot program

Large-scale deployment always led to unforeseen issues and companies can prevent these outcomes by installing their products in a simulated environment in-house. This is carried out through pilot program that simulates customer environment in terms of number of nodes and installed applications.

Let us understand endpoint compatibility model with the help of a case study. The diagram below displays the 4 phases in which endpoint model progresses.

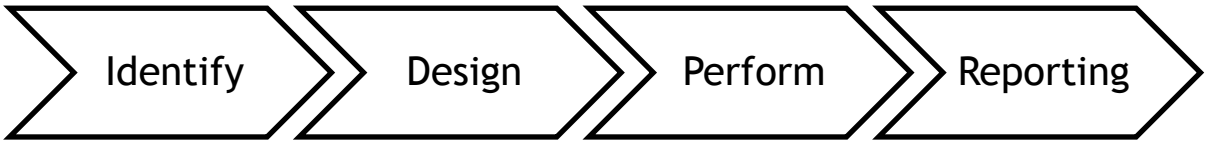


Figure 6 Endpoint Compatibility Model Phases

Identify

The first step in endpoint compatibility model is to identify the endpoint products and their versions that need to be tested. We start with gathering information about environment configuration to test.

Let us take example of two endpoints X and Y managed by a centralized server. X monitors data on the host and alerts the server administrator when any forbidden data is leaked outside the host. Y forbids any malicious file to enter into the host. Although X and Y differ in terms of functionality, they share a common module Z that restricts users logged into the host to tamper with the resources of endpoint X and Y.

Assuming that a new version for endpoint X is released for Windows 7, let us make use of our endpoint model to design the testing strategy for this endpoint X.

Design

Once we identify the endpoints to test, we move ahead with the design phase where with the help of Inter-team collaboration we distinguish the defect identifiers.

Coming back to our case study, as we understand module Z is shared commonly across endpoints X and Y. Inter-team collaboration among endpoint teams X , and Y leads to identify several defect identifiers such as

- DLL's that protect resources

- Watchdog process that accounts for access protection,
- UI component on the server that allows administrator to configure module Z functionality on the client

Based upon these defect identifiers, we design the scenarios to test compatibility between X and Y. Some of the scenarios can include:

a) Install/Uninstall endpoints X over Y and vice versa and verify the integrity of shared DLL files of module Z
b) Tamper the resources of endpoint Y such as registry entries, program files, processes and drivers while endpoint X is installed.
c) Verify that changing UI components to non-default values reflects to client properly while X and Y are present.

Perform

Now that the test scenarios are written and the execution plan is created, we perform the scenarios on the defined test environment.

These scenarios are executed in phases. In first phase, we run test scenarios on an environment with minimal configuration and record the results. In subsequent phases as the product comes near to beta release, we use help of pilot program and try to uncover issues by running it against a simulated customer environment.

Reporting

We capture our results in the test management software. The RAG (Red Amber Green) status is determined. This is a term used to decide a go-no go in various milestones of a product and is used with respect to compatibility in our model.

7. Lessons Learnt

While we did compatibility testing in our product, we identified several areas mentioned below, which we think would help you to follow our framework:

Admit first

Management can praise or be prejudiced about its testing team capabilities but it does not give them the right to judge the issues found on the customer site. We often see that product management does not take responsibility of a customer issue until it is a major road block or customer persists.

We can improve this situation by accepting more proactive strategies. Also, we should learn to accept our product fault that sets the need to adapt compatibility testing.

Prioritize

Remember that 100% testing is not possible, specially, in compatibility; we always deal with modules of other products. Therefore, prioritization of test scenarios is a key that will help a compatibility tester in maximizing the results keeping efforts constant. The factors like checkpoints, Defect Identifiers and 3rd party application categorization and prioritization will always help you to achieve that.

Always Client First

Applications running on client environment are of prime importance to us. Therefore, gather as much information as possible about most installed applications; frequency of OS distribution, number of licenses procured and any other related information and further use this to build a compatibility testing strategy for the three testing types.

Following the above, testing our product on the customer-simulated environment not only will add to product confidence rating but also will build a fine rapport with the clients.

8. Conclusion

The fundamentals of compatibility testing reveal that you will uncover these issues only when your product interacts with other products. Hence, it is difficult to find these compatibility issues via test scripts or regular functional testing. In that case, to mitigate the gaps with other products one needs to follow a specific framework that aids him in finding flaws early in product cycle.

Application Compatibility Framework promises to empower functional testers to perform compatibility testing and guides him to take the pre-defined path to achieve a better result.

We hope that our framework simplifies understanding of application and optimize the efforts put down by testers in finding compatibility flaws.

In terms of results, we found close to 5% of total defects using this framework, which we otherwise would have not gotten in absence of our model. After this approach has been adopted, we also see a downward trend in number of sustaining hotfixes released to the customer due to compatibility issues.

9. Road Ahead

It has been two years since we adopted compatibility testing and we continue to focus on different areas of improvement.

Compatibility in testing context is a niche area and our efforts were focused till now keeping in mind our product requirements.

Though we have observed 10% growth in defects found via our framework from last release, the customer compatibility issues did not show a substantial decline due to several factors such as rapid changes in product portfolio, unavailability of proprietary applications used by customers and increase in number of clients. Currently we have close to 200 Applications in our 3rd Party portfolio that are prioritized and tested as per release requirements.

In future, we plan to take up tasks such as development of Automated Test Bed, combining Interoperability with Compatibility testing, streamlining Identification & Prioritization based upon qualified factors and enhancing OS Architectural knowledge to develop a comprehensive OS based compatibility model.

References

1. http://blogs.computerworld.com/node/3309
2. http://www.phonescoop.com/news/item.php?n=2336
3. http://news.cnet.com/The-XP-alternative-for-Vista-PCs/2100-1016_3-6209481.html
4. http://www.usertesting.com/blog/?p=242?utm_source=e011&utm_medium=email&utm_campaign=Critical%2Busability%2Bmistakes%2Bof%2B2011

Watch Your STEP !

Prabu Chelladurai

Senior Project Manager, Polaris Application Certification Enterprise (PACE)
Polaris Software Lab Canada Inc
5090 Explorer Drive, Suite 401, Mississauga, ON L4W 4M8, Canada
prabu.chelladurai@polaris.co.in

Abstract

Timely delivery of high quality software within budget is no more a nicety; but a necessity. Software testing, with a significant stake in enabling it, compels enterprises to focus on its improvement. However, the volatile nature of today's enterprises triggered by various factors like recession, attrition, technology change, competition and diverse culture impede any improvement effort and make the course and post-implementation of improvement feel more fragile than agile

This paper details a framework named STEP (Software Test Enhancement Paradigm) that provides ample and adoptable guidance towards improving and fortifying software testing. This proven framework has been enriched with best practices from industry standard frameworks like CMMi, TMM, TMMi, and TPI. This paper also outlines nine proven cost effective solutions based on technology and software process [with a case study] to remove fragility and induce agility in the journey of software test improvement. These aspects blended with STEP's unique ability to accommodate 2 flavors of test process improvement: Staged & Continuous and granularly calibrated measurement of maturity enable easier adoption, focused improvement, quicker realization of ROI thereby guaranteed Delivery of Quality !

Biography

Prabu Chelladurai is a Senior Project Manager at Polaris Software Lab Canada Inc and is involved in Software Test Management and Process Improvement for Polaris and its clients. He has managed large testing initiatives; calibrated software processes practiced by Polaris' clients and implemented STEP/TMM practices for the company's testing division named Polaris Application Certification Enterprise (PACE). He is a certified practitioner of CMMI, ITIL and Function Points (CFPS). Prabu has a Masters degree in Software Engineering from Carnegie Mellon University (CMU, Pittsburgh). While at CMU, he has researched and implemented the best practices of Extreme Programming and Scrum in various software projects.

1. STEP Overview

Software Testing holds a significant stake in ensuring high quality. Awareness of the same prompts enterprises around the world to invest a lot in maturing and improving the test processes to ensure their business objectives are met. But are they comfortable in achieving and sustaining high test maturity? Is there adequate focus on all the relevant facets/dimensions of testing? Do they know where their test maturity stands? More often than not, the answer is 'NO'.

STEP (Software Test Enhancement Paradigm) Framework was designed at Polaris Application Certification Enterprise (the testing arm of Polaris Software lab Ltd) to overcome these challenges. STEP Framework abbreviated for Software Test Enhancement Paradigm is business goal driven test process improvement framework. Its features include: focus on five comprehensive test dimensions, granularly calibrated maturity measurement, "Feel Agile. Not Fragile" tools/concepts and the flexibility of two distinct representations. STEP can be implemented as a standalone improvement framework. It can also be implemented in tandem with the likes of Capability Maturity Model Integrated (CMMi), Testing Maturity Model (TMM) and Testing Maturity model Integrated (TMMi) thereby amplifying the benefits. The STEP framework has been represented in figure 1 below:

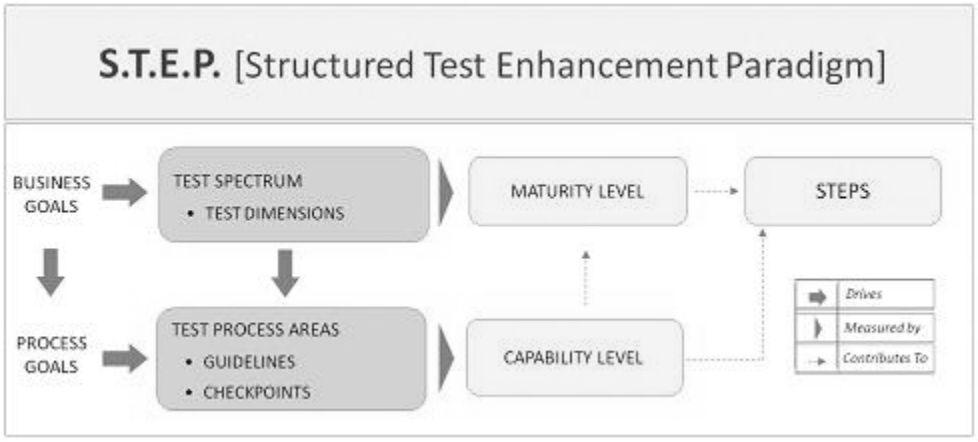


Figure 1, STEP Framework

a. Business Goals: Business goals are the key drivers of the STEP framework. They enable scoping of test spectrum and arriving at various process goals. An example business goal can be: Reduce Cost of Quality by 30%

b. Process Goals: Process goals are usually both qualitative and quantitative and are derived from the business goals. Example: Reduce SIT (System Integration Test) defect leakage by 25%

c. Test Spectrum: Test spectrum comprises of various test dimensions that are considered in scope for the STEP.

- Test Dimensions: Test dimensions are given below (examples of each dimension is given in the brackets '[ ]'):
    i. Type [Functional, Non-Functional]
    ii. Phase [Unit Test, System Test, SIT(System Integration Test), UAT(User Acceptance Test), PVT(Production Verification Test)]
    iii. Technique [Black Box, Grey Box, White Box]
    iv. Mode [Manual, Automated]
    v. Extent [Release Specific, Regression]

d. Test Process Areas: These are the individual units of the STEP Framework

- Guidelines: This is a sub entity of each test process area and provides the various guidelines, suggestions and recommendations for successful implementation of the respective process area. These guidelines can be customized by referring to successful best practices in the industry: CMMi, TMM, TMMi and past lessons learned
- Checkpoints: Checkpoints are sub entities of each test process area which assist application teams in ensuring satisfaction of the process capability requirements

e. Capability Level: This is used to measure the capability of a test process area. Capability level is on a scale of A to D based on the extent to which the particular Process Area is implemented

f. Maturity Level: This is used depict the maturity of the test spectrum pertaining to the Staged flavor of the STEP framework. Maturity level is on a scale of 1 to 5 based on the process areas that have been implemented

g. STEPS: 'STEPS' is a combination of Capability and Maturity Levels. In has a Max Value = 10.00

STEPS = Maturity Level STEPS + Capability Level STEPS
Where,
Maturity Level STEPS = Maturity Level of Testing [Maximum of 5.0 Points]
Capability Level STEPS = Capability Level Points [Normalized to a Maximum of 5.0 Points]

2. STEP Process Areas

The STEP framework comprises of 17 Process Areas that span across all facets of software testing. These process areas were primarily inspired from the Testing Maturity model (TMM) developed by the Illinois Institute of Technology [Illene Burnstein 2002]

1. Test Strategizing
2. Test Specification
3. Test Execution
4. Test Planning
5. Test Monitoring & Control
6. Test Environment
7. Defect Management
8. Test Organization
9. Test Training Program
10. Test Ware Management
11. Test Lifecycle & Integration
12. Verification
13. Metrics Program
14. Knowledge Management
15. Test Process Management
16. Defect Prevention
17. Test Optimization

Each of the above mentioned process areas have their own objective to fulfill. This paper does not detail each of them as the focus is on the overall STEP Framework and its applicability.

3. STEP Flavors

One of the shortcomings in most of the test process frameworks is the lack of flexibility in terms of the options for adoption. STEP Framework targets addressing the same with two flavors: 1. Staged Representation; 2. Continuous Representation. These two representations have been inspired from the widely successful CMMi framework

3.1 Staged Representation

Staged Representation flavor of STEP enables test enhancement in a staged fashion. 17 test process areas are spread across five levels (Maturity Levels). Each of the test process areas identified across the various levels in the framework, have a defined path for staged improvement. There are up to five

maturity levels [Figure 2] available: 1, 2, 3, 4 and 5 with maturity increasing in that order. The staged representation is available in TMM and TMMi

1) Level 1: Initial

This maturity level as the name indicates is where any test organization would start. There are no specific process areas targeted for this level

2) Level 2: Defined & Managed

This level targets at standardizing key testing components that cause an impact at project level. The process areas addressed include: Test Strategizing, Test Specification, Test Execution, Test Planning, Test Monitoring and Control, Test Environment, Defect Management

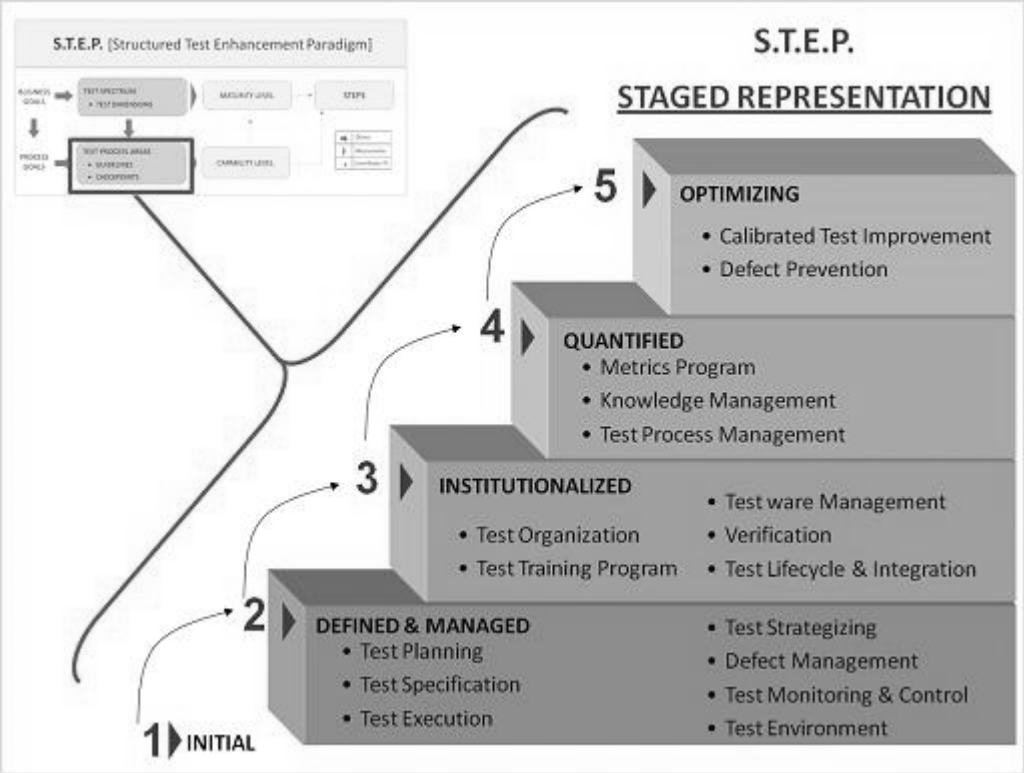


Figure 2, STEP Staged Representation

3) Level 3: Institutionalized

Maturity Level 3 targets institutionalizing the testing practice at an organization level and improving certain engineering aspects of Software Testing. The process areas targeted institutionalizing include: Test Organization, Test Training Program, Test-ware Management, Verification and Test Lifecycle and Integration

4) Level 4: Quantified

The crux of this level is the Metrics program. It also focuses on other process areas like Knowledge Management and Test Process Management

5) Level 5: Optimizing

This is the highest level of maturity focusing on continuous improvement. Process areas like Calibrated Test Improvement and Defect Prevention enable the objective of this maturity level

3.2 Continuous Representation

The test process areas are grouped in to 3 categories to enable flexibility for an organization. Each of the test process areas identified in the STEP Framework, have a defined path for continuous improvement. Each milestone in this path for continuous improvement is called as a Capability Level. This representation that is unique to the STEP Framework allows flexibility in picking and choosing the process areas.

There are up to 4 capability levels [Figure 3] available for each process area: A, B, C, and D with capability increasing in that order. The general criteria to achieve each of the levels are as follows:

- A: Beginners with quality investment (effort, time and money) to achieve goals
- B: Failed to achieve goals with agreeable variance
- C: Goals Achieved with compromises & unknown/new challenges
- D: Goals achieved without compromises, unknown challenges and is repeatable i.e. Feeling Agile!

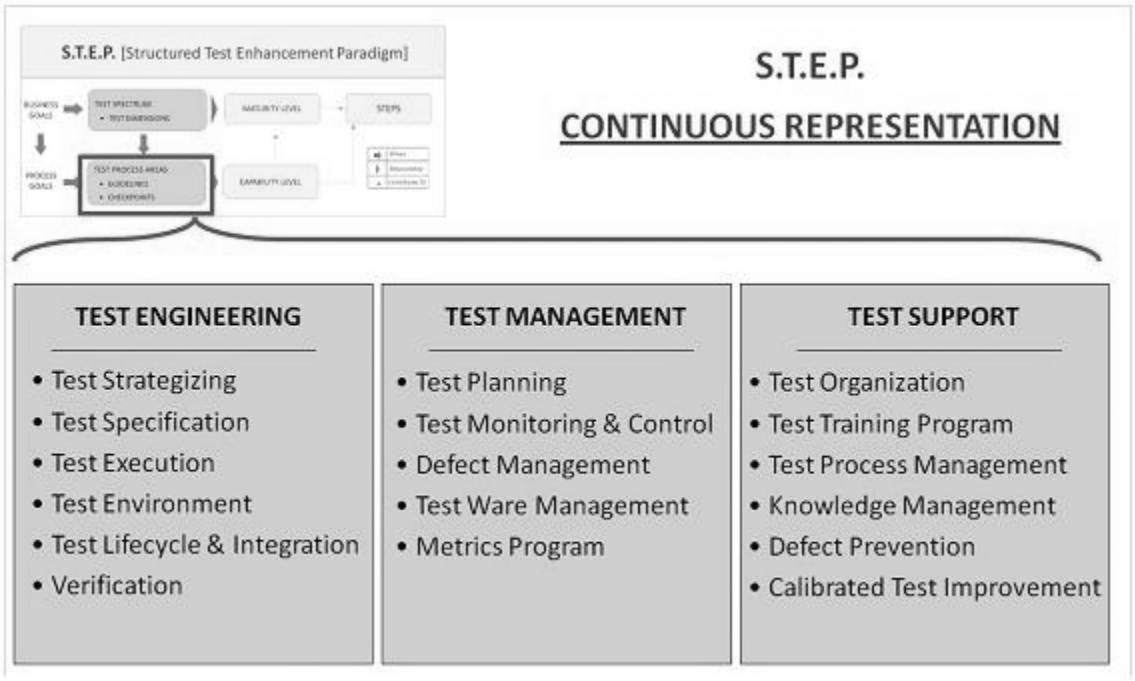


Figure 3, STEP Continuous Representation

1) Test Engineering

This group includes the process areas that focus on the engineering aspects of software testing. The process areas include: Test Strategizing, Test Specification, Test Execution, Test Environment, Test lifecycle & Integration and Verification

2) Test Management

This group targets the managerial aspects of software testing. The included process areas are: Test Planning, Test Monitoring and Control, Defect Management, Test-ware Management and Metrics Program

3) Test Support:
This group constitutes process areas that do not deal with core testing but definitely play a role in enhancing them. The process areas covered are: Test Organization, Test Training Program, Test Process Management, Knowledge Management, Defect Prevention, and Calibrated Test Improvement

The 17 process areas distributed across maturity levels and process Groups in Staged and Continuous representation line up as follows: [Figure 4]

CONTINUOUS / STAGED	TEST ENGINEERING	TEST MANAGEMENT	TEST SUPPORT
LEVEL 1 – INITIAL	-	-	-
LEVEL 2 – DEFINED	• Test Strategizing • Test Specification • Test Execution • Test Environment	• Test Planning • Test Monitoring and Control • Defect Management	-
LEVEL 3 – INSTITUTIONALIZED	• Test lifecycle & Integration • Verification	• Test-ware Management	• Test Organization • Test Training Program
LEVEL 4 – QUANTIFIED	-	Metrics Program	• Test Process Management • Knowledge Management
LEVEL 5 - OPTIMIZING	-	-	• Defect Prevention, • Calibrated Test Improvement

Figure 4, STEP Staged & Continuous Representation

4. Feel Agile, Not Fragile !

The volatile nature of today's organizations is triggered by various factors like recession, attrition, technology change, competition and diverse culture. These factors challenge the ease of achieving and sustaining high process maturity and subsequently hinder the intended objectives of the same. As a result the course of implementation and post implementation of process improvement feel more fragile than agile. This situation would be no different for STEP. Below are 9 practical & proven cost effective ways to remove fragility and induce agility in the journey of software test process improvement.

4.1 'Spectrum'ized Test Strategy

Usually when a strategy is designed with a blank mind there is a lot of room for gaps to creep in. Defining the test spectrum upfront by considering all possible dimensions will definitely help strategize in a structured and quick way. The possible dimensions that apply to most test strategies are as follows (These are the same dimensions that are used by the STEP Framework for configuring the Test Process Areas):

i. Type [Functional, Non-Functional]
ii. Phase [Unit, System, SIT, UAT, PVT]
iii. Technique [Black Box, Grey Box, White Box]

iv. Mode [Manual, Automated]
v. Extent [Release Specific, Regression]

For example, let us assume a test strategy is being developed for an application with the following characteristics.

- The Application has not existed before
- It is internal to the organization
- There will be no more releases or enhancements to this application
- The Application interfaces with many internal applications

The test dimensions to be addressed by the strategy will be as follows:

TYPE	*FUNCTIONAL TEST*		~~Non-Functional Test~~		
PHASE	*UNIT TEST*	~~System Test~~	*SIT*	*UAT*	~~PVT~~
TECHNIQUE	*WHITE BOX*	~~Grey Box~~		*BLACK BOX*	
MODE	*MANUAL*		~~Automated~~		
EXTENT	*RELEASE SPECIFIC TESTING*			~~Regression Testing~~	
Legend >>>	*FOCUSED IN TEST STRATEGY*			~~Not Focused in Test Strategy~~	

Process Areas Addressed:

- Test Strategizing (Maturity Level 2)

Key Benefit(s):

- Structured & Comprehensive Test Strategy with ease

4.2 Verifiable Test Estimation

A Close to accurate estimate is the foundation of successful Test Planning. But achieving it is not an easy task. One way to ensure accuracy of test estimates is to have a secondary estimation technique to verify the estimates arrived at using the primary estimation technique. The common techniques used for Test Estimation are

- *Function Point Analysis*
- *Test Point Analysis*
- *Use Case Point Analysis*
- *Program Complexity*
- *Activity based Estimation*

Depending on the form of requirements at hand it would be prudent to choose a primary and secondary estimation technique from the above list. Creation and use of simple software tools that implement these estimation techniques would simplify the task further

Process Areas Addressed:

- Test Planning (Maturity Level 2)

Key Benefit(s):

- Higher level of confidence in the estimates
- Accurate Test Estimates therefore less chances of budget or schedule overrun
- Quicker and Easier way to compute test estimates

4.3 In-Line Test Automation

Software companies invest in Test Automation. The investment bothers most because of the time it takes to realize the ROI. This often influences siding with Manual Testing alone. But if the Software Lifecycle is

slightly tweaked (to deliver code in iterations), then the ROI on Test Automation can be realized immediately.
Normal Scenario: Test Automation Scripts are prepared for Release 'N' and the same can be executed no sooner than Release 'N+1'
Agile Scenario: If the Code for Release 'N' is released in iterations then, the Automation scripts can be executed as soon as the next iteration in Release 'N'

Process Areas Addressed:

- Test Execution (Maturity Level 2)

Key Benefit(s):

- Quicker realization of ROI on Test Automation
- Significant reduction in Test Cycle time and proportional reduction in Time to Market

4.4 Zero-Effort Test Reporting

Test Reporting is always a critical and time consuming activity. The element of quantitative reporting just complicates things. One of the solutions to feel agile while still fulfill the reporting needs is by automating the process of test reporting. This is not as complex as it sounds. Simple macros that can function independent of the Test/Defect Management Tool are sufficient to achieve this. The diagram below [Figure 5] highlights a proven design that reduced the effort spent in Test Reporting to 'Nil'. *[Ref Appendix 1 for details]*

Process Areas Addressed:

- Test Monitoring and Control (Maturity Level 2)
- Defect Management (Maturity Level 2)
- Metrics Program (Maturity Level 4)

Benefit(s):

- Zero Reporting Overhead
- Better control of the Software Testing Activities
- Easier implementation of Metrics Program

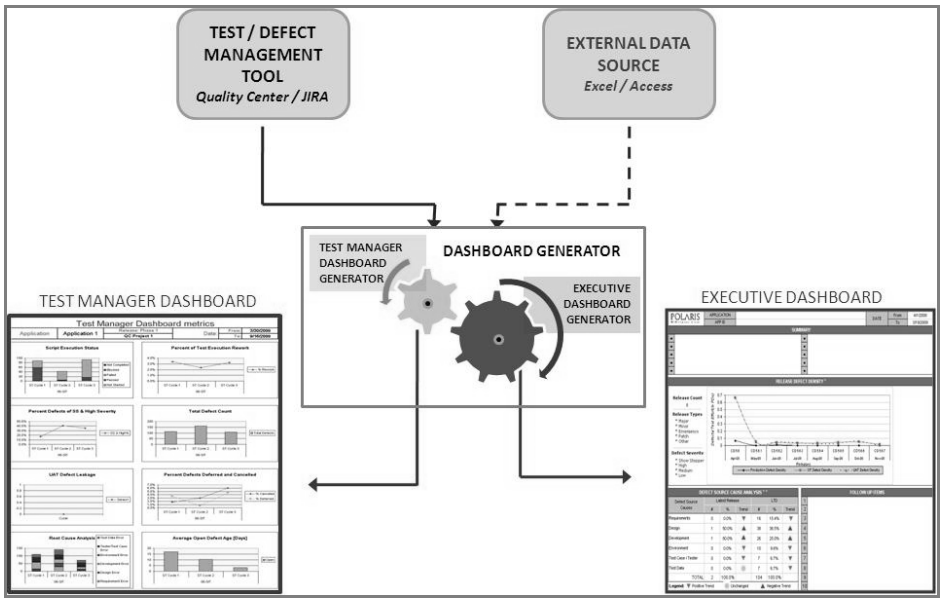


Figure 5, Test Dashboard Generator for Zero Effort Test Reporting

4.5 Sell Internally

If proper awareness is not created, efforts to improve processes are usually seen as more documentation. Hence it is a must to look at innovative ways to inculcate the importance of the process improvement initiatives. In other words, the initiative must be sold to the employees who are the practitioners and not just decided at the Executive level.
For Example, 'Review the Requirements Document'. Employees are usually reluctant to indulge in this. If you can say 10 Requirements Defects have the potential to cost at least USD 50,000, then the seriousness would be understood.

Process Areas Addressed:

- All process Areas (Maturity Levels 2 to 5)

Key Benefit(s):

- Process Improvement will not be looked at as Documentation overhead
- Process will quickly transition to habit

4.6 Test Early, Test Often

When to start testing? Usually testing gets pushed to the end of SDLC. But an ideal solution (in line with V-Model) is to test early and test often. Though this sounds impractical due to the team structure/dynamics, it will really help if 2 goals are pursued: Do More in parallel with development; Thorough Review of Requirements and Design.
Fact: Average Cost to fix one defect [Roger S Pressman 2003] rises significantly through the SDLC

Design Phase	: 1 Hour, USD 82
Construction Phase	: 6 Hours, USD 537
Testing Phase	: 15 hours, USD 1240
Production	: 60 Hours, USD 4959

Process Areas Addressed:

- Test Lifecycle and Integration (Maturity Level 3)

Key Benefit(s):

- Significant Cost and Effort Savings
- Reduced Time to Market and Defect Leakage

4.7 Maximize Coverage <> Maximize Documentation

This practice will definitely help remove the perception that process improvement is about more documentation. One of the ways to accomplish is to reuse test artifacts across different testing phases. This will enable better coverage in later phases. Usually the testing responsibilities of various phases lie with different group of stakeholders. Each group prepares test artifacts from scratch and misses on the coverage aspects instead end up with loads of duplicate test artifacts. If a small contractual change is made to make the test artifacts evolve through phases, then the degree of reuse and coverage automatically increase as well
Process Areas Addressed:

- Test Specification (Maturity Level 2)

Key Benefit(s):

- Effort Savings by Reuse of Test Artifacts across phases
- Enhanced Test Coverage and Reduced Defect Leakage

4.8 Increase Environment Availability

Test Environment best practices usually demand that the test environment must be a mirror of production environment. While this thought cannot be discarded, it is highly important to look at innovative ways to improve the availability and testability of the environments. Availability can be improved by adopting virtualization solutions or by developing harnesses to simulate the application components that are not available.

Process Areas Addressed:

- Test Environment (Maturity Level 2)
- Test Execution (Maturity Level 2)
- Test Lifecycle & Integration (Maturity Level 3)

Key Benefit(s):

- Reduction of impact caused by Test Environment Non-Availability

4.9 Calibrated Test Process Improvement

On a general performance scale, improving from 35% to 50% is much easier than improving from 95% to 97%. Achieving this can be eased if the improvement initiatives at a very high level of maturity are calibrated. One proven solution that can be adopted is the Process Calibration Framework [Figure 6].
The Process Calibration Framework [Muthu Gopal & Prabu Chelladurai 2007] quantitatively analyzes & improves the various aspects of process in lines of factors such as Performance, Stability, Compliance, & Capability using two components: Process Meter and Calibration Tree. *[Ref Appendix 2 for details]*

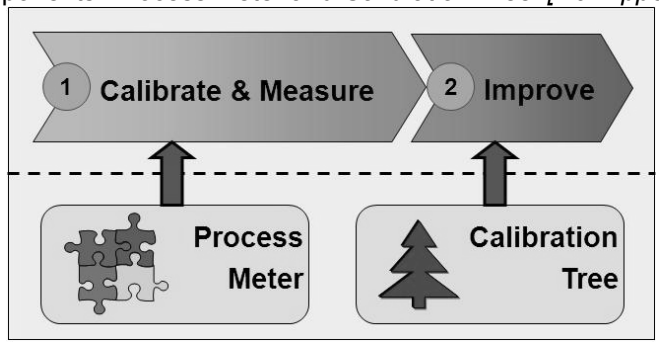


Figure 6, Calibration Framework

Process Areas Addressed:

- Calibrated Test Improvement (Maturity Level 5)

Key Benefit(s):

- Focused and Prioritized Improvement
- Quicker realization of ROI

5. STEP Vs Other Popular Frameworks

STEP framework compares with other popular process improvement framework across many attributes. ***Differences:*** Below is a table [Figure 7] that highlights comparison with some of the key attributes

Attribute	CMMi	TMM	TMMi	TPI	STEP
Exclusive focus on Software Testing	-	X	X	X	X
Flexible Representations for Adoptions	X	-	-	-	X
Continuous (Focus on Process Areas of Interest)	X	-	-	X	X
Staged (well defined evolutionary plateau towards continuous improvement	X	X	X	-	X
Enhanced Coverage with a Robust Multi-Test Dimensional Focus	-	-	-	-	X
Granularly Calibrated Test Maturity Measurement Framework	-	-	-	-	X

Figure 7, STEP Vs Other Popular Frameworks - Differences

Inspirations: Though the above table cites a few differences between STEP and other popular test frameworks used in the industry there are attributes in STEP which have been inspired from these frameworks and refined slightly (as needed). Some of the inspirations from the other frameworks are as follows:

- The concept of Staged and Continuous have been inspired from CMMi
- The 17 Process Areas are primarily inspired from TMM/TMMi. But following changes have been made based on experience in implementing them:

#	Process Area in TMM/TMMi	Equivalent Process Area in STEP	Change Type and Reason
1	Test Design and Execution	Test Specification Test Execution	**[MODIFICATION]** Process Areas have been separated as they have the capability to influence different aspects of Testing
2	-	Defect Management	**[ADDITION]** Process Area has been added as a Process Area as defects directly relate to quality
3	Peer Reviews	Verification	**[MODIFICATION]** Peer Review is a type of Verification and there are many other verification techniques that can be adopted. for example, Inspection, Delphi technique are commonly used review techniques
4	Software Quality Evaluation	-	**[DELETION]** Process Area objectives have been covered as a part of Verification [Level 3]
5	-	Knowledge Management	**[ADDITION]** managing knowlwedge is important for a level 4 company
6	Quality Control	Test Process Management	**[MODIFICATION]** The process area has been renamed to avoid confusion with CMMi Quality Control. Also, the process area has been moved from level 5 to level 4
7	Test Process Optimization	Calibrated Test Improvement	**[MODIFCATION]** The process area has been renamed to place emphasis on the need for calibration

Figure 8, STEP Process Areas Vs TMMi Process Areas

Though STEP can be a standalone test framework, the above inspirations enable it to compliment other popular frameworks and amplify the benefits

6. Case Study

6.1 Institution Overview

The institution under consideration is a division of a global bank that offers transactional services Given below are some facts about the division of the bank:

- Over $276 billion in average liability balances; Over $12.8 trillion in assets under custody; Over $3+ trillion in worldwide transactions daily
- Serving 96% of the world's Fortune 500 companies
- 10 regional processing centers worldwide
- Multiple Domains: Cash Management | Trade Services | Securities & Fund Services
- 93 Software Applications across multiple technologies supporting the above business

6.2 Problem Statement

The IT group of the division under consideration was facing the following problems with respect to Software Testing & Application Quality:

- Increase in Production Incidents
- Increase in support and maintenance costs
- Increased unplanned releases and patches
- Ineffective & Ad hoc Testing

Ultimately resulting in,

- Increased Cost of Quality
- Increased Time to Market
- Decreased Customer Satisfaction

6.3 STEP Solution

Upon an analysis of the problems and consideration of the business and process goals it was decided to adopt ***'Continuous Representation' of the STEP Framework with focus on Test Engineering and Test Management***. In lines with the decision, following process areas were chosen for improvement:

PROBLEM AREAS / PROCESS AREAS	Increase in Production Incidents	Increase in support and maintenance costs	Increased unplanned releases and patches	Ineffective & Ad hoc Testing	Increased Test Turnaround Time
Test Strategizing	X	X	X	X	X
Test Planning					X
Test Monitoring and Control					X
Test Specification	X			X	
Test Execution	X		X		
Test Environment		X	X		
Defect Management				X	
Metrics Program		X	X		
Test Training Program	X			X	
X - Indicates the which Problem Area(s) were primarily addressed by the Process Area					

Figure 8, Problems addressed by Process Areas

Maximum STEPS possible is 10.00. Considering only a few processes were chosen, the Maximum STEPS for the identified processes is 4.65. The number of STEPS in the AS-IS Process was found to be 1.66. Details of the analysis are as follows:

Maximum STEPS = 10.00

Max STEPS for Process Areas considered = 4.65
- *Capability Level STEPS* = *2.65*
- *Maturity Level STEPS* = *2.00*

STEPS in AS-IS Process = 1.66
- *Capability Level STEPS* = *0.66*
- *Maturity Level STEPS* = *1.00*

Following are the key activities performed in order to achieve the Target STEPS:

- Identify goals for the process areas identified for improvements.
- Clear set of guidelines and checklists were defined based on best practices from CMMI, TMM, TPI and past learning.
- Following 'feel Agile, not fragile' concepts were implemented targeting some of the process areas considered in scope:
  - 'Spectrum'ized Test Strategy
  - Verifiable Test Estimation
  - Zero Effort Test Reporting
  - In-Line Test Automation
  - Sell Internally
  - Maximize Test Coverage not Documentation

After 6 months of pilot implementation and 12 months of a steady-state implementation, the Target STEPS (4.50) were achieved except for a couple of process areas. Details of the calculation are as follows [Figure 9]:

STEP CALCULATION		AS-IS SCORE		Target Score		Actual Score	
Capability Level & Capability Level Points	· Test Strategizing	A	1	D	4	D	4
	· Test Specification	A	1	D	4	D	4
	· Test Execution	B	2	D	4	D	4
	· Test Planning	B	2	D	4	D	4
	· Test Monitoring & Control	A	1	D	4	D	4
	· Test Environment	A	1	D	4	D	4
	· Defect Management	A	1	D	4	D	4
	· Metrics Program	-	0	D	4	C	3
	· Test Training Program	-	0	D	4	C	3
CAPABILITY STEPS		0.66		2.65		2.50	
MATURITY STEPS		1.00		2.00		2.00	
STEPS		1.66		4.65		4.50	

6.4 Benefits

Figure 9, STEP Calculation for Improved Process

The benefits reaped speak volumes for success of the implementation

- 70% Reduction in Sev 1 Outages [Figure 9] which enabled reducing support costs by 21%
- Additional potential annual savings of USD 500,000 via Automation of Test Metrics (for 75 Applications)
- 34% reduction in unplanned releases
- Streamlined & Robust Test & Defect Management Processes
- Around 15% Reduction in Maintenance & Support Costs

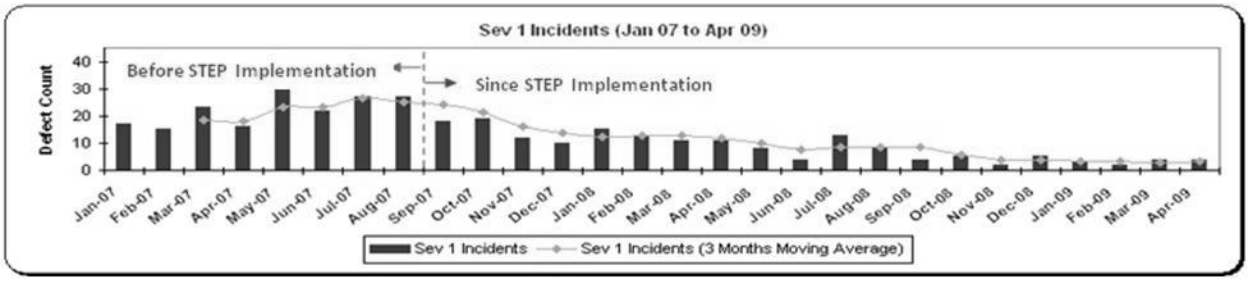


Figure 9, Sev 1 Defects from Jan '07 to Apr '09

Appendix

Appendix 1: Test Dashboard Generator

The Test Dashboard Generator advocated as a part of STEP includes the following features

- Comprehensive set of Test Metrics
    - Script Execution Status
    - Test Execution Rework
    - Percent of Severe Defects
    - Total Defect Counts
    - Defect Leakage
    - Percent Defects Deferred
    - Percent Defects Cancelled
    - Root Cause Analysis & Trend
    - Defect Age
    - Defect Density
- Dashboards for Senior Executives & Test Managers/Leads
- 100% End-to-End Automation
- No Separate License Costs
- Blends well with 'Any' Test Management Tool
- Extensively Customizable

Given below is a sample Test Manager Dashboard generated using the Test Dashboard Generator:

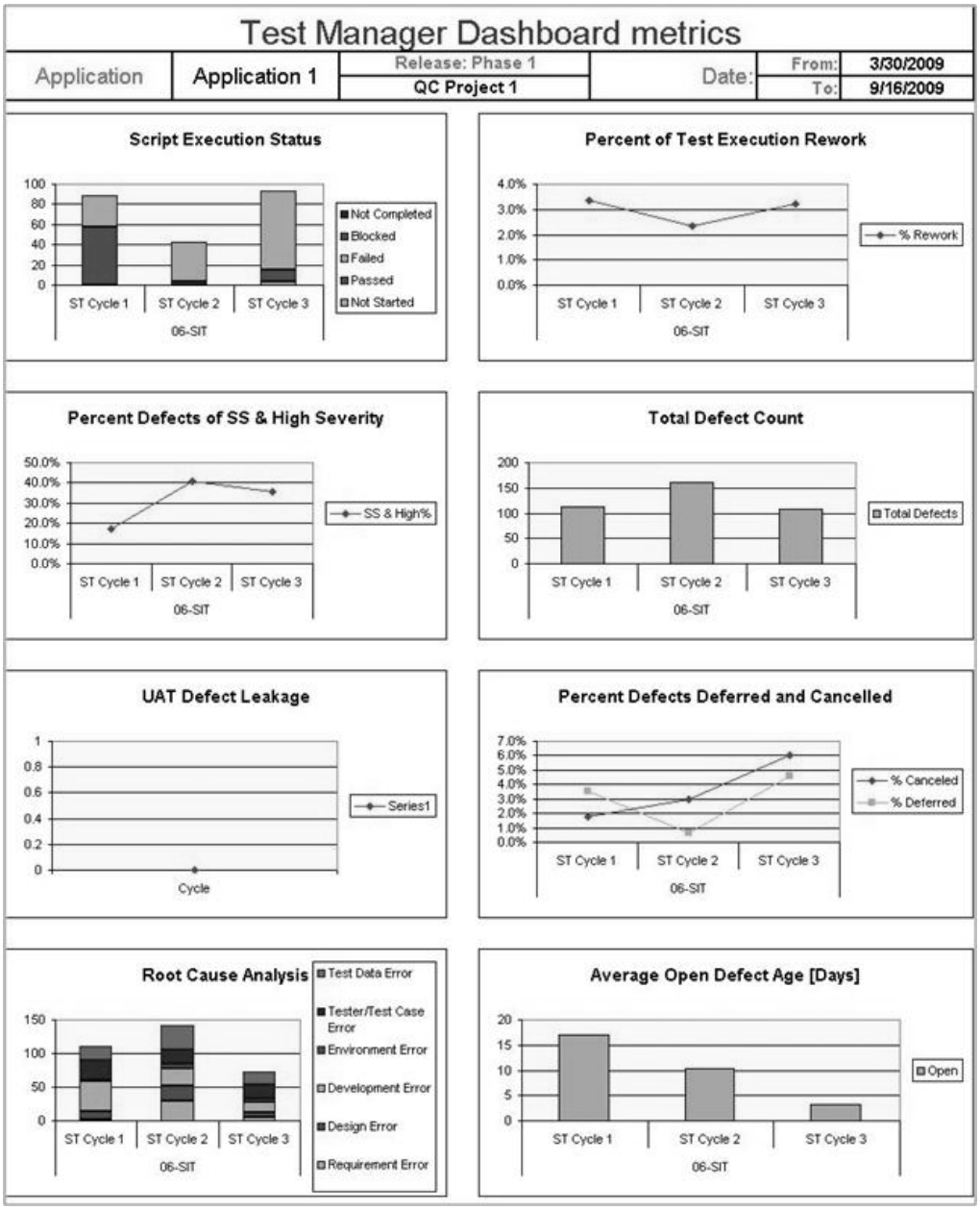


Figure 10, Sample Test Manager's Dashboard

Appendix 2: Process Calibration Framework

What is Calibration? - Calibration in general is associated with instruments and is defined as the act of aligning the instruments to a measurement unit(s) to enable it to respond in terms of the unit(s). For example calibrating a thermometer to Fahrenheit scale means the thermometer can be used to read temperature in terms of Fahrenheit. **What is Process Calibration?** Extending the definition of Calibration, Process Calibration can be defined as aligning the process to certain calibration factor(s) so that the health of the process can be measured with respect to the calibrated factors(s). **The Process Calibration Framework** quantitatively analyzes & improves the various aspects of process in lines of factors such as Performance, Stability, Compliance, & Capability using two components: Process Meter and Calibration Tree.

The process of calibration happens in 2 phases:

1. Calibrate and Measure [using Process Meter]

   This Phase involves the following Steps:

   - Identify Sub-Processes
   - Identify Calibration Factors. Some common factors include
     - *Performance* [The degree to which quantitative goals (quality, quantity, cost, time) of the process are met]
     - *Compliance* [The measure of cross conformance between the process and the requirements]
     - *Stability* [The consistency of the measurable attributes (quantitative goals) of a process]
     - *Capability* [The ability of the process to cater to the requirements]
   - Associate suitable Test Metrics with the identified calibration factors
   - Assign Ranks [0 is the minimum rank and 100 is the maximum rank]
   - Compute Process Health [0 is the minimum rank and 100 is the maximum rank]

2. Improve [using Calibration Tree]

   - The calibration tree has the score in the process meter as its root. Any score that needs to be analyzed forms the root of the calibration tree.
   - The score branches out into the attributes, which were responsible for the score on the process meter.
   - The reason behind the value of the attribute(s) is analyzed in lines of the 3As: Actors, Artifacts & Activities. This analysis helps interpret the strengths and areas of improvement in the process

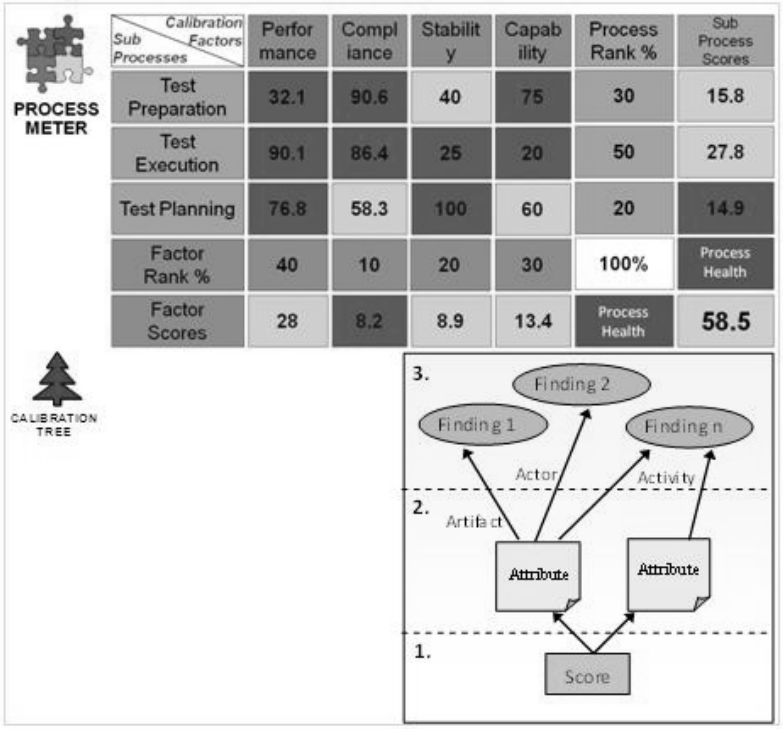


Figure 11, Process Meter and Calibration Tree

The above diagram (figure11) shows the calibration analysis performed on 3 testing process areas. The overall health is 58.5 on a scale of 100. This score is generally considered low in the calibration scale and hence can be improved a lot. In order to improve the overall score, we will have to target improvement of the weaker areas of the process. A more careful analysis of the process meter will help identify these weak areas. In this example, the performance of the test preparation process, the stability and capability of the test execution process are the key reasons for the low overall score. Now these identified areas have to be passed up the Calibration Tree to narrow down on the exact improvement effort

References

[1] Mary Beth Chrissis, Mike Konrad, Sandy Shrum. CMMi®: Guidelines for Process Integration and Product Improvement – 2nd Edition. Addison-Wesley Professional, 2006.

[2] Burnstein, Ilene. Practical Software Testing: A Process-Oriented Approach. Springer Professional Computing,2002.

[3] Muthu Gopal, Prabu Chelladurai. Calibrate Your Estimation Process presented. 2nd Annual International Software Estimation Colloquium by QAI, 2007

[4] Roger.S.Pressman. Software Engineering, A Practitioner's Approach - 5th Edition. McGraw Hill, 2003

[5] Eric Van Veenandal. Test Maturity Model Integration. www.tmmifoundation.org, 2010.

Reverse Engineering: Vulnerabilities and Solutions

By
Barbara Frederiksen-Cross (barb@jli.com) and
Susan Courtney (susan@jli.com)

ABSTRACT

The same characteristics that provide for cross-platform deployment of many modern software development languages also renders the software written in these languages extremely vulnerable to reverse engineering. At the same time, reverse engineering tools and techniques have become much more sophisticated. The convergence of these two developments creates substantial risk for software developers with respect to both the security of their software and protection of trade secrets and intellectual property that is embodied in the software.

Fortunately, the risks that reverse engineering poses to your intellectual property, competitive edge, and bottom line can be mitigated if you take proactive measures to protect your software against reverse engineering.

This paper first examines the ways in which software is vulnerable to reverse engineering and then explains techniques that may be incorporated into your software quality program to help protect your software assets against reverse engineering. The paper also discusses factors that must be considered and weighed when deciding which anti-reversing techniques to apply.

BIOGRAPHY

Barbara Frederiksen-Cross is the Senior Managing Consultant for Johnson-Laird, Inc., in Portland, Oregon.

Susan Courtney is a forensic software analyst and has worked as a consultant for Johnson-Laird, Inc.

1 Introduction

This paper examines the areas in which software is vulnerable to reverse engineering, and explains techniques that may be used to help protect your intellectual property against reverse engineering practices.

The idea for this paper was inspired by the needs of a client who experienced first-hand the reverse engineering of their software and its proprietary communication protocols. The software in question was vulnerable to automated reverse engineering techniques because it did not employ effective anti-debugging and anti-tampering measures. As a result of these weaknesses, competitors reverse engineered both the operation of the software and a proprietary encrypted communication protocol that was used to control the client's hardware products. The information was used to create competing products, permit copying of proprietary content, and preempt control of the client's devices.

2 Disclaimer

While it is possible to hinder reverse engineering in a number of different ways, no software technology can completely prevent a program from being reverse engineered. Whether reversers are ultimately successful in reverse engineering well-protected software depends ultimately on factors such as their skill, motivation, and the amount of time, money, and technology they are willing to devote to the effort.

Anti-reversing techniques are used to obstruct reversers by making the process of reverse engineering extremely difficult, slow, and expensive, thereby discouraging all but the most persistent, skilled, and well-funded reversers.

Multiple factors must be considered when deciding which anti-reversing techniques to apply. Every anti-reversing approach has an associated cost that must be measured in terms of its impact on system performance, code reliability, implementation ease, deployment considerations, maintenance requirements, product testing and production debugging issues. The selection and effectiveness of anti-reversing techniques must also consider details specific to the software and the operating environment that it uses.

3 Reverse Engineering Overview

Reverse engineering is the process of working with byte-code[1] or compiled code to gain an understanding of a program's logic and data. Programmers will recognize at least some of the reverse engineering techniques discussed in this paper because many reversing techniques use common debugging tools for a familiar purpose – to reveal the exact operation of the software in a controlled set of circumstances. In addition to common debugging tools, there are also a variety of special purpose reverse engineering tools that can help a reverser automate this process.

Programs written in certain languages are inherently more vulnerable to reverse engineering than others. With programming languages/frameworks such as Java and .NET, an intermediary product, often called byte-code, is shipped to the customer instead of compiled binary code. These frameworks provide for platform-independent code development by running the byte-code version of the software within a virtual machine environment that hides the details of the hardware upon which the software is running. The virtual machine manages the program's execution environment including interaction with the underlying hardware and operating system so that the program need not be aware of platform-specific differences. For the byte-code program to run on a particular computer, the virtual machine environment

[1] Byte-code is a generic term used to describe intermediary output. In Java, the intermediary output is known as Java bytecode. In .NET, the intermediary output is known as MSIL, which stands for "Microsoft Intermediate Language."

on that computer compiles the byte-code to the machine-readable instructions required for that specific hardware platform.

Byte-code based frameworks are more vulnerable to reverse engineering because byte-code based languages rely on an intermediary form of program instructions that are often very close in appearance to the original source code. A savvy programmer can inspect unprotected byte-code to reveal a program's inner workings, or use special software to recreate source code that can be used (illegally) as the basis for a competing product.

The development frameworks for languages such as Java and .NET also contain many pre-coded base classes that support common programming tasks such as user interface, network connectivity, mathematic functions, cryptography, data access, database connections, and services required for web-based applications. Programmers writing software in these frameworks build their applications using these base classes in combination with the code they create. Base classes are well-documented, and use of such building blocks (when visible in the byte-code or executable) provides additional clues that help the reverser understand the operation of the code. These clues may also help the reverser isolate specific functions of interest within the code.

Reverse engineering is most often accomplished with the aid of specialized software tools such as debuggers, disassemblers, and decompilers that permit the reverser to examine the sequence of instructions that comprise a computer program.

The tools and techniques used by reversers fall into two broad categories: dynamic analysis, which is analysis of the software as it is running, and static analysis, which is performed against a stored copy of the executable software. Thus, countermeasures used to prevent reverse engineering fall into two primary categories: techniques used to prevent dynamic analysis of the program while it is executing and techniques used to prevent static analysis of the binary code.

3.1 Dynamic Analysis Tools

3.1.1 Debuggers

The most common tool for dynamic analysis is called a debugger. Debuggers are designed to interact with programs and their runtime environments to capture information about unexpected behaviors ("bugs") that might be encountered during software testing. But debuggers also provide would-be reversers with the ability to inspect the memory, data, and files the program is using, and to allow the reverser to trace the sequence of operations within a program while it is running.

Debuggers allow a user to monitor and control the environment of the program as it executes. By using breakpoints[2] to pause the program's execution when specific criteria are met, control can be surrendered to the debugger software. At that time, the user can examine the contents of the program's environment, including data stored in the program's memory and the contents of files or file output areas. Debuggers also allow the user to step through the program one instruction at a time as the program is running ("single stepping")[3] and to capture and log various events and information as the program runs.

Most debuggers monitor the program from a very low-level perspective, capturing interactions between the program and the computer's processor, memory, and operating system. Program logic is generally represented in both machine language and assembly language formats. Even modestly complex programs may include millions of low-level machine or assembly language instructions, so

[2] Breakpoints can be set based on a variety of criteria, such as execution of specific instructions, access to specific memory locations, invocation of particular programming interfaces, or access to specific files.
[3] This process lets the reverser control program execution so that each individual instruction the program issues can be intercepted and examined.

reverse engineering an entire program start to finish by using a debugger is typically a tedious and time-consuming process.

A skilled reverser often reduces the volume of code that needs to be reverse engineered by exploiting her knowledge of the points at which an application program interfaces to the Windows operating system or to the user. By analyzing the program's use of interfaces, a reverser may be able to quickly locate the specific processing operations of interest within the targeted program. As a simple example, if the target program prompts the user to enter an activation key, and a reverser wishes to see the test required to validate the key, they can do so using debuggers such as Ollydbg, Shadow, WinDbg, or SoftIce. The reverser first searches for the error message sent to the user when an invalid key has been entered. This technique quickly localizes the area of the program that performs the processing of interest. By working backward from the point where the error message is issued the reverser can identify where in the code the activation key is evaluated, what tests are used to determine a valid key, or where to make modifications that will permit the key test to be bypassed entirely. By setting a breakpoint at the user prompt, the reverser can replay the prompt/response/validation code sequence as often as necessary to confirm her understanding.

Similarly, if a reverser can identify key data variables, file accesses, or memory locations, she can set breakpoints that are triggered by access to these locations, and use access or modification of these items as the starting point in her analysis. Once a breakpoint is reached, the program pauses and the reverser can inspect both the currently executing instructions and any associated data values in memory. This technique is often helpful to identify processing of proprietary data transformations, including encryption and decryption methods.

3.1.2 Dumpers

Dumpers are programs that typically capture the contents of a program's working memory at a specific point (or points) in time while it is running. This copy of memory is saved to disk for subsequent analysis. Most dumper programs also capture and save additional information that relates to the environment in which the program is running and information managed by the Windows operating system that concerns the program's operational state. Examples of memory dumpers include programs such as Explorer Suite and LordPE. Many dumpers also incorporate some of the dynamic capabilities common in debuggers.

By analyzing a memory dump, a reverser can examine the data the program was using at the time the dump was taken. Further, the reverser can also inspect all portions of the program that are present in memory at the time the memory dump is captured. The reverser can extract the memory values to a separate file for subsequent analysis. Once extracted, the reverser can use specialized software such as a disassembler to analyze and translate any portion of the program that is not encrypted, thereby rendering the captured memory contents as a series of human-readable instructions or data values.

3.1.3 Virtual Machines

Virtual machines ("VMs") are software implementations of real hardware. A VM imposes a layer of abstraction between the running program and the computer's hardware and operating system. This abstraction layer provides a controlled environment in which any program can be run as though on the real hardware, but under the watchful eye of the VM software. Virtual machines are sometimes used in the context of dynamic analysis so that a reverser can save periodic snapshots of the program state and operating environment as she attempts to reverse engineer the program. If the program terminates for some unexpected reason, these snapshots can be used to start the reverse engineering session again at some predetermined point, so that the reverser does not need to start over from the beginning of the run. The use of virtual machines can also help a reverser defeat some forms of anti-debugging protection.

3.1.4 Other Tools Used In Dynamic Analysis

Reversers also employ registry and file monitoring software during dynamic analysis. A registry monitor can be used to intercept registry keys that may be used to store application information. This tool facilitates the recovery of license keys or decryption keys, if they have been stored in the Windows registry.[4]

A file monitor can be used to trap accesses to all files used by the application. This analysis can help the reverser understand the program's operation. It also helps the reverser identify files that may be used to hide a security algorithm or encryption key so that this information may be intercepted.

3.2 Static Analysis Tools

3.2.1 Disassemblers and Decompilers

Disassemblers and decompilers are used to analyze a static copy of the program stored on disk. Generally speaking, a disassembler reads a machine-readable version of a program, parses it to identify the discreet sequences of instructions that make up the program, translates these instructions to a human-readable form, and writes the translation to a file for later analysis.

The output of a disassembler is usually low-level assembly language source code that provides a human-readable representation of the program at a much lower level of abstraction than the original source code.[5] Most debuggers include disassembly capabilities while other products, such as IDA Pro, are directed more specifically to the task of disassembly, and may include special capabilities to optimize the resultant translation.

Because the assembly code version of a program may be millions of lines long, including instructions for very common functions such as displaying windows and accepting user input, reversers seldom attempt to reverse engineer the totality of the program. Instead, the reverser searches the converted code for a particular character sequence (such as a message, literal value, or exported function name) to use as a starting point, and then works backward or forward from that point to locate desired algorithms or functions in the program. Once located, the reverser can analyze the specific sequence of instructions to understand how they operate.

Decompilers and reverse compilers are similar to disassemblers. However a decompiler reads in executable code, assembler code, or byte-code, and writes out source code in a higher level language, such as C or the original programming language used to write the program. With the exception of decompilers used for byte-code languages, the resultant code is often very different from the original source code, though much easier to read than low-level assembly language.

Products such as SourceAgain, JReversePro, JODE and Jdec produce good Java source code from .class files. Products such as Reflector and Reflexil perform similar functions for .NET code, allowing a reverser to extract very complete MSIL (.NET byte-code) that may be nearly identical to the original source code.

---

[4] The windows registry is a special directory that stores settings and options used for the Microsoft Windows operating system. It also contains information relating to hardware and user-specific settings. Many programs store information such as license keys and initialization settings in the registry.

[5] Note: This model does not hold true for development platforms such as Java and .NET. Disassemblers for these platforms read in byte-code and may produce output that is very similar to the original source code unless the byte-code has been obfuscated.

3.2.2 Other Tools Used In Static Analysis

Tools such as hexadecimal editors ("hex editors"), unpacker programs, and file analyzers are also useful when performing static reverse engineering. A hex editor allows the reverser to search and view a human-friendly representation of compiled programs. The hex editor may be used to run initial searches for textual elements, such as error messages, that can help identify portions of the program that are of particular interest. A hex editor may also be used to provide a human-readable view of binary code for visual inspection and analysis.

An unpacker program may be used to unpack compressed or encrypted code so that it can be processed with a disassembler or viewed with a hex editor. Many unpackers also include features that help to remove certain types of formulaic obfuscation.[6]

File analyzers are used to identify the compiler/packer/obfuscator that was used with the software of interest, so that the appropriate decompiler/unpacker/deobfuscator can be used.

4 Thwarting the Reversers

The primary techniques to prevent dynamic analysis include page guarding, debugger detection, and virtual machine detection. The primary techniques used to prevent static analysis are encryption and obfuscation. These techniques will be discussed respectively in this following section.

Techniques used to prevent static and dynamic analysis can be implemented directly in the program code as it is written, or they can be applied after the entire program has been written via the use of third-party software. In the latter case, the third party software reads in the source code or executable version of the software, performs a series of modifications to implement the desired protection techniques, and writes out the transformed source or executable. Applying protection after development is completed minimizes the impact on the development lifecycle.

4.1 Preventing Dynamic Analysis

Dynamic analysis is difficult to defeat because program instructions must be unencrypted to execute. This means that there will always be some portion of the program that is exposed to inspection by debugging tools or memory dumpers. Debuggers and memory dumpers can help defeat encryption by permitting the reverser to capture the portions of a program that are unencrypted in the computer's memory during execution. Once the areas of the program that handle encryption are identified, dynamic analysis can also be used to obtain decryption keys that are stored in memory and to locate and analyze decryption algorithms by tracing the program's instructions during execution.

4.1.1 Debugger Detection

To minimize a program's vulnerability to dynamic analysis, a program must first be able to detect that it is being run in a debugging environment, and then once detected, to take some form of defensive action. The tests for debuggers may be coded as a part of the program, or the protection may be applied via automated tools that provide a secure environment wherein the program runs. A general scenario for debugger protection is described in the following paragraphs.[7]

---

[6] Obfuscation is the generic name for a variety of techniques that are used to reduce a program's vulnerability to static analysis. Obfuscation techniques modify the program's symbols, internal data, layout, and logic in a way that preserves the function of the program but makes the program's operation much more difficult to decipher and understand.

[7] A complete discussion of debugger detection is beyond the scope of this paper. Detailed information is available online *at http://www.openrce.org/reference_library/anti_reversing* (last visited 2011-08-04)*;*

On startup, the code to detect debuggers loads first, and prevents the protected program from loading if it detects that a debugger is in use. If no debugger is present, the protected program is allowed to load and begin execution. As it runs, the protected program periodically relinquishes control back to the debugger detection software to ensure that the runtime environment is still safe. If a debugger is detected while the program is running, the debugger detection software initiates a pre-determined defensive action.

Possible defensive actions include immediately stopping the program, changing the behavior of the program so that the reverser is led to an erroneous understanding of its logic, and altering the program to force some eventual error after wasting the reverser's time.

4.1.2 Considerations for Debugger detection

Techniques to detect debuggers include inspecting specific debugger status information maintained by the operating system when a debugger is running, such as checking the names of running programs to identify any known debuggers, and creating a process that looks for evidence that debugger software is installed on the computer's hard disk.

Some debuggers create breakpoints by modifying the programs they monitor. These debuggers inject interrupts into the target program after it is loaded into memory. This activity can be detected by monitoring or scanning the program memory for these changes (a process called *check summing*). A related technique, called *page guarding*, seeks to interfere with the creation of breakpoints by preventing or healing such changes to the program.

Since the use of a debugger typically slows a program's execution, it is sometimes possible to detect the presence of a debugger by computing the amount of time required to run specific program instructions. Once computed, this value is tested against a predetermined threshold to see if the execution is slower than expected. This test is predicated on assumptions about how long the tested instruction should take to run in a particular environment, so this technique can only be used if environmental information is available before the program is shipped to customers.

The techniques used for debugger detection all require a solid understanding of both the operating system and the various debuggers available for the specific platform. Technical information required for debugger detection is subject to change with new releases of either the Windows operating system or new debugger software. Due to the specialized knowledge and skills required for debugger detection, the most pragmatic solution is often to add debugger protection via the use of commercial products, especially when this protection is needed for legacy software products.

One downside of debugger detection and defensive action is that the logic required to perform debugger checks uses system resources and therefore may affect performance. With certain techniques for debugger detection, it is possible that a user with a legitimate debugging session unrelated to the target program might trigger the defensive actions, causing the protected program to terminate unexpectedly.

To address these issues, most commercial products used to apply debugger detection allow the user to select among options that offer varying levels of protection.

4.2 Preventing Static Analysis

Encryption and obfuscation are the primary techniques used to prevent static analysis. These techniques may be used independently or combined together.

---

http://www.symantec.com/connect/articles/windows-anti-debug-reference (last visited 2011-08-04); or the four part anti-debugging series located at *http://www.veracode.com/blog/?s=anti-debugging* (last visited 2011-08-08).

4.2.1 Encryption

Encryption uses a mathematical formula to transform a program's byte-code or binary code into a form that is no longer comprehensible to the reverser or to automated disassembly software.

One approach that interferes with static reverse engineering is to use internal self-encryption of the program's binary code. A program that is self-encrypting is stored in encrypted form and only decrypted during execution. Using self-encryption protects a program because comprehensible instructions are only exposed to inspection during the comparatively short time when the program is actually running. This technique, sometimes called *packing*[8] or *binary encryption*, will prevent static analysis and automated disassembly, unless the reverser can find some way to first decrypt the program – i.e. through use of a specialized unpacker program or repeated memory dumps.

For encryption to be effective the target program and the decryption/encryption routines used to protect it must both be encrypted. In some cases, the decryption routine is designed to decrypt the entire program into memory at the start of execution. This form of encryption is extremely vulnerable since the entire unencrypted program can be read from memory (using a debugger or memory dumper) while the program is running.

Alternatively, a better technique called just-in-time encryption ("JIT encryption") decrypts only a small segment of the program as it is needed during execution, and then re-encrypts it before the next segment is decrypted.[9] This technique offers additional protection against attempts to capture an unencrypted version of the program as it runs.

So-called polymorphic encryptors may use multiple encryption algorithms and unpacking modules within the same program to render detection and reversal even more difficult.

4.2.2 Considerations for Encryption

Encryption techniques help protect against static analysis, but do not directly address dynamic analysis because instructions must be decrypted before they can be run. Encryption is most effective when only small portions of the program are decrypted into memory at any particular time. This technique prevents the use of dumper software to capture an unencrypted version of the entire program from memory.

Unfortunately many third-party packers and binary encryption products themselves have been reverse engineered. If a reverser can identify the specific packing/encryption tool used, an automated solution to assist with unpacking or decrypting the protected code may be readily available. To avoid this possibility, encryption techniques may be accomplished using custom software. This approach provides some additional security but at the expense of developing, deploying, and maintaining the custom encryption software.

Whether encryption software is purchased or custom written, there is always a potential down-side in terms of performance. From the user's perspective, software that is decrypted in its entirety at run time may appear to take longer to start. Software that uses JIT encryption will run slower than an

---

[8] Many varieties of packer software exist. Historically, packers were used to compress the size of executable code as an aid to distribution and storage on secondary media. A compressed executable includes the original program code, which has been compressed to reduce its size, packaged along with the relevant decompression code routines in a single executable. Running a compressed executable requires first loading the decompression routine to uncompress the original program code and then transferring control to the uncompressed code. Today, many packers include at least rudimentary capabilities to perform encryption and obfuscation.

[9] Just-in-time encryption should not be confused with just-in-time compilation, a different technique that is used to improve performance for programs written in bytecode-based languages such as Java.

unprotected version of the same software. How much slower is determined by the way that encryption is implemented and the specific processing performed by the decryption and encryption routines.

The impact of encryption on performance can be tuned to some extent by encrypting only selected portions of the overall software. The performance of JIT encryption is also sensitive to the size of the segment being decrypted, which can often be selected at the time encryption is added.

4.2.3 Obfuscation

In addition to encrypting, software may be obfuscated to help defeat static analysis. Obfuscation makes a program more difficult to understand and attempts to thwart automated disassembly tools.

In addition to transforming the program code to hide the original logic, an obfuscator may rearrange the internal organization of the software and add irrelevant code blocks ("garbage code") which serve to increase the apparent complexity of the program and lead reversers away from the true program logic. Some obfuscation techniques also introduce code changes that prevent automated disassemblers and decompilers from producing an accurate rendering of the program code.

Obfuscation is a generic name for techniques used to reduce a program's vulnerability to reverse engineering. Obfuscation techniques fall into three general categories: lexical transformations, control flow transformations, and data transformations. At best, obfuscation makes it time-consuming, though not impossible, to reverse engineer a program. The various types of obfuscation techniques, their effectiveness, and technical considerations are discussed below.

4.2.4 Lexical Transformations

Lexical transformations remove textual clues that reveal how the program operates making it harder for a reverser to *zero in* on specific algorithms and processing routines of interest. Lexical transformations can be used to replace program identifiers, such as the names used to identify program data, and function names that identify the program's processing routines. Lexical transformations may also be used to disguise textual content such as error messages or user instructions.

Lexical transformations are useful because they make the program harder to read and understand, but offer only a weak protection when used alone. Modern disassemblers include sophisticated tools such as call path analysis and visual representations of code structure. These tools are designed to help a reverser overcome the lack of meaningful names and symbols.

4.2.5 Control Flow Transformation

Control flow transformations seek to defeat disassemblers by exploiting weaknesses in the way the disassembler translates binary code into assembly language instructions. Control flow transformations may alter the order and flow of the program logic to make the logic harder to follow.

For example, some control flow transformation techniques add or modify conditional and branching instructions that complicate the program. In this type of modification the test results for the artificial branch condition must be calculated at the time the program is run, so a disassembler cannot know which branch would be taken and must therefore disassemble both. The obfuscator that transforms the software actually knows how the condition will be evaluated in advance, and structures the inserted branch so that when the program runs it will always take the same logic path that was followed in the original code.

Other control transformation techniques modify the actual "jump" instructions that control the logic flow in the binary code. In an unmodified program, the jump instruction often provides the address of the next instruction to be executed. To obfuscate this control flow, the program is modified to force it to compute the address for the target value of that branch. This technique prevents a disassembler from

determining the logic flow because a vital piece of data, the address of the next instruction, is not known at compile time.

Another technique used for obfuscation is to break up the logical order of a program, making it harder to follow. In this technique, high level code structures, such as functions, are replaced with less transparent abstractions. For example, a complex function such as processing a request from a user is broken into multiple components which are scattered throughout the program, perhaps interleaved with other code that serves some other purpose within a single routine. Once functional routines are broken up and dispersed, the control of the program is modified to ensure that these components are still executed in the proper order. This type of change can add significant complexity to the control flow but still preserves the original function of the program.

4.2.6 Data Transformation

Data transformations serve to disguise important data structures within the program by altering the way data is stored or represented within the program. Examples of techniques used for data transformation include modifying or splitting table structures, replacing literal values stored in the program with computations that deliver the same value, encrypting textual error messages and literal values, and dividing important structures into multiple pieces that are not all stored in the same way.

4.2.7 Combined Techniques

Many of the techniques described above may be combined to further strengthen their effectiveness. For example, the human-readable code may be compiled normally, and then further processed to replace all unconditional branches with conditional branches plus false predicates. Computations are inserted for “true” branch targets, to prevent detection of the true target by the disassembler. Targets of the false branches are pointed to positions before the real target, which are padded with bad or misleading code.

4.2.8 Automating Obfuscation

Obfuscation transformations may be applied to software using either manual or automated processes and to either the source code or the binary translation that is produced by a compiler. Performing manual obfuscation transformations can be very time consuming, thus obfuscation is commonly made using obfuscator software. There are many different obfuscator programs available in the marketplace, both as commercial products and as shareware. Most offer a variety of protective transformations, and give the user some level of control over what specific techniques are to be applied.

In a typical scenario, the target program is compiled, and then the executable binary code is input into the obfuscator along with parameters that tell the obfuscator which types of obfuscation are desired. The obfuscator transforms the code and writes out the obfuscated version of the software that will be made available to customers.

4.3 Considerations for Obfuscation

Obfuscation is most effective when combined with encryption, and it is normally best practice to use a combination of obfuscation techniques so that lexical content, control flow, and data structures are all transformed or obscured. The benefits of automated obfuscators include ease of use, ongoing support by a third party whose focus is automated obfuscation, and minimal impact on code development and maintenance. One drawback: most obfuscation techniques provide protection against static analysis, but do not directly address dynamic analysis. Obfuscation benefits must be weighed against their effect on system performance.

Transformations of lexical content and data structures are primarily directed to making a program harder to understand for a human reverser. Transformation of lexical content does not normally carry any performance penalty but it provides only modest protection against reverse engineering. Transformation of data structures usually has a comparatively small effect on performance. While it provides a greater hurdle than simple lexical transformation, it does not prevent the use of automated disassembly tools.

In contrast, transformation of logic flow can be very effective in defeating the automated tools used for static analysis, but these transformations may significantly affect performance. Depending on the specific techniques used, logic flow transformations can also make the program much harder for a human reverser to understand.

In addition to merely increasing program complexity, it is important to choose an obfuscator that produces transformations resilient to automated removal. Obfuscation transformations may increase the size of the program, its memory requirements, and its demand for processing resource.

Further, reversers understand that software developers may employ obfuscation tactics. Because many obfuscation transformations are somewhat formulaic, reversers may use deobfuscator programs[10] to attempt to restore the programs original structure. As the name implies, deobfuscator software is designed to help reversers analyze an obfuscated program by removing encryption and obfuscation transformations. The deobfuscator does so by attempting to detect and reverse the formulaic transformations applied by obfuscators.

5 Conclusion and Recommendations

None of the techniques used to protect against reverse engineering can guarantee that software will never be reverse engineered. That said, there are anti-reversing techniques that can make the task of reverse engineering more difficult and more time consuming. These techniques can be applied in one of two ways: they can be added as modifications to the program source code, or they can be applied to the executable version of the program before it ships to customers.

All of the anti-reversing techniques discussed in this paper can be added to a body of software via the use of automated software tools. These anti-reversing techniques may be applied to the entirety of the software or to selected portions. As a minimum, we recommend that the following protections:

- Encrypt the executable version of the software, preferably with JIT encryption
- Use a combination of obfuscation techniques to provide the software with additional protection against reverse engineering
- Don't forget to extend the same protection to software or firmware upgrades

A variety of products are available to automate the addition of anti-reversing techniques. These tools vary in sophistication, cost, techniques employed, ease of use, and effectiveness. The selection of automation tools should be predicated on an understanding of which techniques provide a best fit for your security goals, performance requirements, and cost sensitivity. A comparison of several of the most common products is included in Appendix A.

Finally, you may want to consider enhancing your legal protection to help you in the event your software is compromised. Although these measures cannot prevent reverse engineering, they can provide a tactical advantage to help you seek remedies in the event a breach does occur. Some of the options available to help you in this regard include strengthening the terms of your end user license agreement, filing a deposit copy of each version of your software with the US Copyright Office, and obtaining patent protection for any novel aspects of your product.

[10] Many common obfuscators have themselves been reverse engineered, revealing their specific techniques, algorithms, and encryption strategies. This information has been used by reversers to create automated deobfuscation software that can be used to attempt to reverse the effects of obfuscation. Such deobfuscators are also referred to as "unpackers."

6 Appendix A: Feature Comparison

This chart summarizes feature and price information for a sample of nine alternate software protection products (including two different Dotfuscator versions). This sample is by no means exhaustive, but rather reflects the capabilities of some of the more popular or more robust offerings in this market.

Pricing key: $ cost range < $1000, $$ cost range $10K -$20K, $$$ cost range $50K-$100K+

	Dotfuscator Community Edition	**Dotfuscator Professional Edition**	**Cloakware Security Suite**
Supported Languages	.NET	.NET	Most compiled languages, no .NET
Encryption	N	some	Y
JIT Decrypt	N	N	Y - whitebox
HW locking	N	N	
Lexical Obfuscation	some	Y	Y
Control Flow Obfuscation	N	Y	Y
Data Obfuscation	N	Y	Y
Debugger Detection	N	some	Y
VM detection	N	N	?
Page Guarding	N	Y (SW, via checksum + custom method)	Y
Tamper checking	N	Y	Y
Integration Consulting	N	Y	Y
Licensing Model	Bundled with MS Visual Studio	Base +Per Developer	Annual license or per device
Cost	Free	$$	$$$
Hooks	Post CIL	Post CIL	Source and post compile
Time to deploy	Days	Days to 2 weeks	Est. 2 months
Notes	Not recommended as this offers only weak protection	Has recently added instrumentation platform which includes use analysis	

	StrongBit ExeCryptor	9Rays Spices Obfuscator	Silicon Realms Armadillo (Passport)	Oreans Themida & Code Virtualizer
Supported Languages	Most compiled, no .NET	.NET and managed C++ assemblies	Most compiled languages, no .NET	Most
Encryption	Y	String only	Y	Y
JIT Decrypt	Y - see note	N		Y – see note
HW locking	Y		Y	?
Lexical Obfuscation	?	Y	Via encryption	Y
Control Flow Obfuscation	Y - code morphing	Y	Via nanomites	Y – code morphing
Data Obfuscation	?	Y	N	?
Debugger Detection	Y	N	Y	Y
VM detection	?	N	N	Y
Page Guarding	?	N	Y	Y
Tamper checking	?	Y	Y	Ring 0
Integration Consulting	?	?	Custom build available at no charge	?
Licensing Model	?	Per build machine or enterprise	Per developer	Annual license
Cost	$	$$	$	$
Hooks	Post compile	Post CIL	Post compile	Post compile
Time to deploy	Days	Days to 2 weeks	Days	Days to 2 weeks
Notes	Very unique approach to obfuscation. Protected SW morphed and portions run in embedded VM. Partners with 9Rays	Partners with StrongBit	Custom build is recommended, offered at no additional charge	Code translated and executed via embedded emulator. Unpopular with Reverse Engineers

	Arxan TransfornIT	v.i.Laboratory CodeArmor
Supported Languages	Most compiled languages, lighter coverage for .NET (no repair guard)	Most compiled languages, separate version of product for .NET
Encryption	Y	Y
JIT Decrypt	Y	Y
HW locking	Y	
Lexical Obfuscation	Y	Via encryption
Control Flow Obfuscation	Y	N
Data Obfuscation	Y	N
Debugger Detection	Y	Y
VM detection	Y	Y
Page Guarding	Y – multi level w/ guards and repair guard	Y
Tamper checking	Y	?
Integration Consulting	Y	Y
Licensing Model	Perpetual, Tiered, or per app	Tiered, per app
Cost	$$$	$$ - $$$
Hooks	Post compile	Post compile/Post CIL
Time to deploy	Days to 6 wks	Days
Notes	At least 3-5% overhead, this assumes guards tuned	At least 2-4% overhead

Dirty Tricks in the Name of Quality

Ian Dees
ian.s.dees@tektronix.com

Abstract

We join software projects with grand ideas of tools, techniques, and processes we'd like to try. But we don't write code in a vacuum. Except on the rare occasions when we're starting from scratch, we're confronted with legacy code we may not understand and team members who have been quite productive for years without the silver bullets we're pushing.

How do we get a toehold on a mountain of untested code? How can we get our software to succeed despite itself? Sometimes, we have to get our hands dirty. We may have to break code to fix it again. We may have to put ungainly scaffolding in place to hold the structure together long enough to finish construction. We may have to look to seemingly unrelated languages and communities for inspiration.

This introductory-level talk is a discussion of counterintuitive actions that can help improve software quality. We'll begin with source code, zoom out to project organization, and finally consider our personal roles as contributors to quality.

Biography

Ian Dees was first bitten by the programming bug in 1986 on a Timex Sinclair 1000, and has been having a blast in his software apprenticeship ever since.

Since escaping Rice University in 1996 with engineering and German degrees, he has debugged assembler with an oscilloscope, written web applications nestled comfortably in high-level frameworks, and seen everything in between. He currently hacks embedded C++ application code, automates laboratory hardware, and writes test scripts for Tektronix, a test equipment manufacturer near Portland, Oregon.

When he's not coding for work or for friends, you're most likely to find Ian chasing his family around on bicycles, plinking away at his guitar, or puzzling at the knobs on the espresso machine while some impromptu meal simmers on the stove nearby.

Ian is the author of Scripted GUI Testing With Ruby *and co-author of* Using JRuby*, both published by the Pragmatic Programmers.*

1. Introduction

We just finished a big project at work. At times like these, it's natural to reflect on the practices that got us to this point, to see which ones are effective and which ones we should abandon. As we engaged in this sorting activity, a third category emerged: practices that helped in a surprising way. For want of a better term, I'm going to refer to these as *dirty tricks*.

What's a dirty trick? It's certainly not a universal best practice.[1] It's also not an antipattern, though. Antipatterns are near-universal worst practices; for some examples, see Moss Drake's "Sabotaging Quality" paper (which he's presenting here at PNSQC 2011).

No, dirty tricks sit somewhere between those two extremes. They're practices that may help in one specific context, and be a terrible idea in other contexts. They may only be useful as a desperate last resort, when the alternative is doing nothing and watching your software—or your career—slowly stagnate.

So, why am I bringing up these practices today, if they're so context-specific or such potentially bad ideas? There are a few reasons to talk about dirty tricks:

1. Because dirty tricks may only be valid in a narrow context, they're a good reminder to consider the context of *any* proposed tool, technology, methodology, or practice. The next time you hear someone tout a so-called "best practice," you can remember this talk and engage your skepticism gear.

2. You may actually find one or two of these techniques useful as a kind of digital dynamite if you're on a project whose progress has become dammed up with untamed code.

3. I hope to elicit a few groans of recognition and keep you entertained for the next hour or so.

This is primarily a talk on software construction. So we're going to start off with some code examples of dirty tricks. Since many of us program in C++, that's the language I'll use for the examples. From there, we'll zoom out one level, and look at ways to shoehorn testing into a reluctant project or subsystem. Finally, we'll talk about more touchy-feely topics like survival in the workplace.

In short, this is a talk on *mindfully* taking on a little technical debt, with our eyes fully open to the costs and consequences.[2]

2. Assorted Dirty Tricks

2.1. The Meta-Dirty Trick

The first and most important tip we're going to discuss, and something I actually consider to be a good practice in most contexts, is:

Don't get fired.

As I've said, the activities we'll be discussing are not best practices. They're not even recommendations. They may be a terrible idea in your context or on your project. Don't undertake them heedlessly. For that reason, this is the only tip you'll see stated in the imperative. The rest of the dirty tricks are all nouns or noun phrases, just to drive the point home.

---

[1] As if there were such a thing.

[2] For more on technical debt, see http://www.martinfowler.com/bliki/TechnicalDebt.htm.

2.2. Blunt Code

2.2.1. The Code Crowbar

The code crowbar is a blunt instrument for cracking open a project and getting a little testing in. Like a regular crowbar, this digital version is meant for one-time use—eventually, we'll discard it and install a proper entry point for our tests.

This dirty trick comes in handy with legacy code that has no unit tests. The proper way to add testing may be to change the design dramatically. But that's risky and takes time. With the crowbar, one can add a tiny bit of testing—just one or two methods in a single class, for starters—and then use those tests to drive the improved architecture.

For example, consider the following C++ class:

```
class Widget
{
public:
    // ...

private:
    int vim;
    int vigor;
    int Pizazz() { return vim * vigor; }
};
```

Let's imagine the `Pizazz()` calculation is a little less trivial and more error-prone than the simple multiplication shown here. We'd like add some tests for it, like this:

```
class WidgetTest
{
public:
    void TestPizazz()
    {
        Widget w;
        w.vim   = 2;
        w.vigor = 5;
        assert(w.Pizazz() == 10);
    }
};
```

All of the `Widget` fields and methods we're using are private, so this code won't compile. The real issue is that we're dealing with a bunch of separate things that really belong together—they're all part of the class's configuration. A permanent improvement will probably involve moving those private properties into a completely new configuration class. If this is a much larger class in a more complicated system, that task might require lots of changes to the code. Breaking up a big module with no tests is risky business.

The crudest way possible to get the above test code to compile and run is to add the following definition just before we `#include` the definition of the `Widget` class:

```
#define private public
#include "widget.h"
```

Horrible, isn't it? We shouldn't do things this drastic lightly. We can do better than this. We could *(temporarily!)* give our test code access to those internals by adding the following line to the `Widget` class:

```
    friend class WidgetTest;
```

This kind of intrusive monkeying with a class's internals does not make for long-term quality. But it gives us a toehold, so that we can move this configuration information to its own class:

```
class WidgetConfig
{
public:
    WidgetConfig(int vim, int vigor)
        : vim(vim), vigor(vigor) {}

    int Pizazz() { return vim * vigor; }

private:
    int vim;
    int vigor;
};
```

…and replace those private fields in the `Widget` class:

```
class Widget
{
public:
    // ...

private:
    WidgetConfig config;
};
```

Now, our test code doesn't need to resort to dirty tricks anymore:

```
class WidgetConfigTest
{
public:
    void TestPizazz()
    {
        WidgetConfig wc(2, 5);
        assert(wc.Pizazz() == 10);
    }
};
```

This example illustrates one property common to several dirty tricks: what we did was temporary. We started with getting the tests in by any means necessary, we did the hard work to improve our code quality, and then we cleaned up our mess.

2.2.2. Macro Abuse

Developers have an allergic reaction to C++ preprocessor macros, and with good reason. Search stackoverflow.com for the term "macro abuse" for some rather entertaining examples of macros making spaghetti code even worse. Many uses of macros can be replaced with more type-safe constructions like new-style casts, templates, and so on.

However, the preprocessor is there for a reason. It's there for you to write "code that writes code," albeit in a somewhat crude fashion. Imagine the following hypothetical code based loosely on a real-life example:

```
someComplicatedFunction(LONG_NAME_FOR_FOO, WITH_FOO_AND_BAR, BAR_VALUE);
someComplicatedFunction(LONG_NAME_FOR_QUUX, WITH_QUUX_AND_BAZ, BAR_VALUE);
```

Did you spot the probable error? If we look closely at the names of the macros, we can see that the first line deals with the names `FOO` and `BAR`. The second line looks like it was meant to deal with `QUUX` and `BAZ`. But the end of the line refers to `BAR_VALUE`, which is likely a typo or a cut-and-paste error. It was supposed to be `BAZ_VALUE`.

If we define a couple of helper macros:

```
#define DO_SOMETHING_WITH(k, v) {\
    someComplicatedFunction(LONG_NAME_FOR_ ## k,      \
                            WITH_ ## k ## _AND_ ## v, \
                            v ## _VALUE);             \
}
```

…we can avoid that particular cut-and-paste error entirely:

```
DO_SOMETHING_WITH(FOO,  BAR);
DO_SOMETHING_WITH(QUUX, BAZ);
```

You're all seasoned pros who know how the preprocessor works. And yet, I bring this example up because of all the horror stories that make us feel guilty for using macros at all. It's okay to use blunt instruments sometimes.

With this foundation of guilt-free coding established, we can build on it with a couple of related tips that lean on the preprocessor.

2.2.3. Testing? Testing!

Ideally, the code that runs in our unit tests should be the same as the code that runs in the production system. Sometimes, though, things get more tangled. The code we're testing might link to a function in a completely different subsystem that has a bunch more dependencies. Those dependencies will presumably have their own tests.

If we're in the process of writing a unit test, we don't want to be forced to drag a bunch of other subsystems in. For example, imagine we encounter the following call in the middle of the `Aim()` method of a `MindControlLaser` object:

```
if (systemBatteryLevel() < LOW_BATTERY_THRESHOLD)
{
    // logging code
}
```

The first unit test we write for the `Aim()` function is presumably concerned with making sure the aiming algorithm is correctly implemented. The logging and battery systems aren't part of this class's contract. The right thing to do in the long term is decouple the `MindControlLaser` from any specific battery implementation. There are lots of ways to do this, as we'll see later on in "The Stubmarine."

Those big refactorings will have to wait until we've got some tests in, though. Let's temporarily excise the problematic code from our unit tests:

```
#ifndef TESTING
//... system battery level stuff from above
#endif
```

We'll set the `TESTING` flag when we compile our unit tests, but leave it unset when we build the production app. Later, we can refactor the code and remove this temporary scaffolding.

2.2.4. Static Assertions

Another useful application of the preprocessor is *static assertions*. These are actually a broadly useful technique, but I'm still bringing them up here as a dirty trick because nearly all plain C implementations of the idea rely on some form of source code cuteness.

Static assertions document assumptions about the program that the compiler can verify. For instance, suppose we're working on an application to route salespeople to cities. Most parts of the system—the user interface, the storage engine, and so on—don't care what the maximum number of salespeople is that we can support. So we've got a constant defined in our program somewhere like this:

```
#define MAX_SALESPEOPLE 100
```

But let's say we have a super-fast routing algorithm that's only proven correct with a sales staff of fewer than a thousand people. We'd like to be able to document this property in the code:

```
STATIC_ASSERT(MAX_SALESPEOPLE < 1000)
```

If a well-meaning coder (perhaps even us!) comes along later and changes `MAX_SALESPEOPLE` to 2000, the compiler will complain and they'll be forced to confront the issue: either replace the algorithm with a slower one that works on larger data sets, or decide that the limit is something we can live with.

Jon Jagger has written a nice catalog of compiler tricks used to implement static assertions.[3] The most reliable of these relies on a case statement:

```
#define STATIC_ASSERT(condition) \
    switch(condition) { case 0: case condition: break; }
```

As long as the condition is something the compiler can evaluate to true or false, this code will test your assumptions at compile time.[4]

2.2.5. The Wheel, Reinvented

"No code is better than no code"—Ezra's Law[5]

It's always better to use an off-the-shelf library than to write your own, isn't it? I'd rather reach for a library or something out of a common tools folder to do just about any common task—decoding JPEGs, etc.—than spend hours writing my own buggy version. Those hours should be spent delivering features to paying customers.

And yet… there are times when it's okay to reinvent the wheel. If you have quality or legal concerns about a piece of third-party code, and if the task you're undertaking is extremely well-defined, then the risk of rolling your own solution is low.

---

[3] http://www.jaggersoft.com/pubs/CVu11_3.html

[4] The upcoming revision of the C++ standard has an official mechanism for these kinds of assertions.

[5] http://www.longbeard.org/2008/07/13/Ezras-Law.html

Test frameworks are a good example of things you might want to write yourself. They're pretty easy to do. They're not impossible to get right. Heck, they're practically a rite of passage in some programming languages.[6] If you're concerned about getting Legal to approve an open source license, or just don't want to put up with someone else's idea of what goes in `main()`, give it a shot.

How easy is this? A test "framework" can be as simple as an agreement to use the built-in C `assert()` macro and name your test methods `Test...()`. For a more useful example of a minimalist framework, see John Brewer's MinUnit exercise.[7]

2.3. Test Practices

2.3.1. The Stubmarine

A moment ago, we looked at an example that, in just a few lines, entangled the otherwise simple `Aim()` function of a `MindControlLaser` class with a specific battery system:

```
if (systemBatteryLevel() < LOW_BATTERY_THRESHOLD)
{
    // logging code
}
```

For the moment, we slapped an `#ifndef TESTING` around those lines. But now, let's dig in and see what this code tells us about our design. Why does the `Aim()` function need to check the battery level? Is it just something someone left in for debugging purposes? Is it to provide a cue for the operator? Is it part of the contract of this class that it checks the battery before aiming?

Once we've answered those questions, we'll know more about what our tests need to do. If the battery check was just something we had in there for debugging, an cheap way to make this function more testable is to turn the battery level function into a pointer:

```
if (batteryLevelFunc &&
    (*batteryLevelFunc)() < LOW_BATTERY_THRESHOLD)
{
    // logging code
}
```

The testing code can leave this function set to `NULL`, while the development version of the app can point it at the real battery function—or even selectively enable/disable the check at runtime.

If it's actually part of the specification for `MindControlLaser` that the `Aim()` function does something specific with a low battery level, we'd like to be able to test that. To do so, we can provide a stub function that returns canned values:

---

[6] Such as Ruby. And SQL, apparently; see http://chrisoldwood.blogspot.com/2011/04/you-write-your-sql-unit-tests-in-sql.html

[7] http://www.jera.com/techinfo/jtns/jtn002.html

```
double fakeLowBatteryLevel()
{
    return LOW_BATTERY_THRESHOLD / 2.0;
}

void testAimWithLowBattery()
{
    MindControlLaser l;
    l.batteryLevelFunc = &fakeLowBatteryLevel;
    l.Aim();
    assert(l.WarningLightOn());
}
```

Of course, if the thing you're replacing is a C++ class, you've got even more options—such as having your real battery and fake battery implement the same interface.

For more sophisticated behavior (e.g., recording whether or not a function gets called), you may look into a more full-featured mocking library like Google Mock.[8]

2.3.2. Cut and Paste

An engineer confronted with a crooked picture will "buy a CAD system and spend the next six months designing a solar-powered, self-adjusting picture frame while often stating aloud [the] belief that the inventor of the nail was a total moron."[9]

Sometimes the right thing to do when faced with a problem is just fix it quickly. Earlier in "Macro Abuse," we saw how cut and paste led to a subtle error, and how doing tricks with the programming language helped us avoid the problem. Now just for fun, we're going to talk about the opposite side of the coin.

There are a few times when a simple cut and paste is easier, faster, and even more reliable than whipping up a Rube Goldberg contraption to write our code for us. One of these cases you'll frequently encounter in the C++ world is makefiles.

When we first introduce testing into a project, we may find ourselves needing two radically different groups of build settings: one for the real program, and one for the tests. These settings may include compiler flags, linker settings, and perhaps even different compilers outright.

If the differences are big enough, the crude "everything's a global constant" approach of makefiles may require you to write a bunch of spaghetti in your build rules. The simplest solution here may be two makefiles. It's either that, or try to use the weak abstraction mechanisms offered by most build systems. The cure may be worse than the disease.

2.3.3. Wranings and Erors *[sic]*

This next item is barely even a dirty trick at all. It's helpful in more situations than our other dirty tricks, and its use isn't even frowned on all that often. The practice: telling the compiler to complain more loudly.

This technique takes several different forms, depending on the programming language. C++ has the `-Werror` compiler flag to treat all warnings as errors you must fix before continuing. Perl has `Test::NoWarnings` and `use warnings`, among other things.

---

[8] http://code.google.com/p/googlemock

[9] http://www.netfunny.com/rhf/jokes/96/Oct/engineers.html

Why is this important? In a big build, it's easy for a compiler warning to slip by unnoticed in the log (or be buried in your IDE). More often than not, the warnings I've seen have indicated real bugs in the code. There have been a few false positives over the years, but these all had easy workarounds.

I'm a big fan of this approach, but still consider it a slightly dirty trick because it affects your teammates as well. If someone's legitimate programming technique happens to be caught by the compiler, they're going to question the value of all this extra checking—and justifiably so. Fortunately, time heals most objections, as more developers see the stricter settings catch a bug that otherwise would have made it into production.

2.3.4. Manualization

Manualization means making something manual. I'm using it here to mean the opposite of automation. More specifically, I'm talking about setting up something manual that has some of the same desirable properties as an automated process.

Consider the practice of continuous integration, where a build server is constantly pulling the latest updates from source control and rebuilding the project. These often run an automated test suite as well.

A large, complex product may have a suite of automated tests that takes hours to run. With source code checkins happening every minute, such a long turnaround time cancels a few of the benefits of continuous integration. We may throw more hardware at the problem, or run the tests less often, or run only a certain subset of the tests on the continuous integration server. But there's another possibility as well. We can make integration and testing a side effect of the normal build.

If you have a large enough team of developers, you can perform something akin to manual continuous integration. Several times throughout the day, people can pull the latest code from one another and build it on their own. They can test by hand on their own systems. They may not even consciously be testing all the features that have changed. But in the act of working on their own specific task, they may notice when something unrelated has gone wrong.

Eric Raymond has said on multiple occasions, "Given enough eyeballs, all bugs are shallow." We might offer a corollary: "Given a few eyeballs less, many bugs are shallow."

2.3.5. Inspiration from Elsewhere

Inspiration can come from surprising places.

C++ programmers have been doing automated unit tests of their code for decades. There are libraries, IDEs, seminars, and books that help with this task. And yet… a lot of developers haven't tried incorporating this practice yet. They may be working in domains where these kinds of unit tests are less helpful. Or they may just not have seen enough value in the practice to push them past the initial stage of the learning curve.

There are other programming communities, though, where having code-level tests is practically a given for any project. There are both cultural and technological reasons for this. If every library you download has a test directory and the language itself comes with a test framework, new developers feel more encouraged to write tests for their own code.

A related practice is refactoring: changing the design of an existing program piece by piece, without breaking it. The Smalltalk folks made this practice famous, but you don't have to be a Smalltalk developer to use the techniques. I finally began to gain a toehold of understanding in this subject when the book *Refactoring: Ruby Edition* came out. Seeing the examples in the language I use for fun (Ruby) made it much easier to apply the concepts in the language I use at work (C++).

2.4. Social Beings

2.4.1. TPS Reports With No Cover

Ever seen the movie *Office Space*? The main character, a programmer, gets told off by manager after manager for failing to use the new, presumably improved, cover sheet for his TPS report. The episode was meant to point out a dysfunctional organization. But there's a second lesson in there as well. Sometimes the "official" format for documentation is a good thing. At other times it slows you down.

In my talk at PNSQC last year, I told the story of a brainstorming session with our team's software lead, where we typed straight into our text editors and generated a preliminary PDF of the requirements for a particular feature. That PDF was the first thing we checked into source control. Sure, the requirements weren't duly marked, numbered, and annotated—yet.

Think of it as a kind of "documentation debt." We knew we were eventually going to convert the document into the official format with the required template. But if we'd started there, we'd never have gotten off the ground.

2.4.2. Playing Hooky

More work time equals more work done—sounds logical, right? And yet, we need to recharge. We need to be able to change directions mentally. Spending all of our time, up to the last minute, on a single urgent project brings diminishing returns.

What else is there to do during office hours besides work? How about going to the occasional conference? Since we're all attending PNSQC as we speak, I'm assuming you're all receptive to the idea that a few days spent outside the lab once in a while are a good thing.

By slacking off and going to hang out with people who inspire us, we learn more about our craft. We sharpen ideas. We gain renewed energy for attacking the great problems that await us back in the office.

2.4.3. The Heist

We spend most of our workday chained to one main computer. The machine is sufficient for day-to-day coding, and then…. Suddenly we need a server for a quick experiment. Or a machine that can run OmniFocus. Or a computer with PhotoShop installed. Or a laptop we can try four different Linux distros on for compatibility testing.

Any office that's big enough will have dark corners with unused computers lying around. Maybe it's a hand-me-down from someone who just got an upgrade, or a VM someone spun up in the server farm weeks ago for testing.

We shouldn't let good hardware go to waste. If we need computing power, we should feel free to ask for it and exercise it. Notice I said, "ask." This is one of those rare times in the talk when I'll use the imperative voice. Please don't steal a machine someone was about to use, or make people wonder where missing hardware went.

It's okay to fancy ourselves as scrappy, resourceful underdogs, as long as we're above board about the whole thing. We communicate, communicate, and communicate again. We find out who owns the box, and let them know how long we'll be using it. We leave a big giant Post-It™ note in its absence. We give it back as soon as anyone needs it, or when our experiment comes to an end.

2.4.4. Job Transformers

How many of you here work at a fairly large company? A lot of us, it looks like. There are a few things big organizations can do to help a software developer. Even the most cynical cubicle dweller can appreciate the ability of a large shop to get lab space and expensive hardware.

Of course, we all know the downsides of working in cubeville: bureaucracy, heavyweight process, lack of tool choice, and lack of autonomy, just to name a few. These aren't unique to big companies, and there are a few shops that seem to be able to escape these pitfalls as they scale. But the stereotype exists for a reason.

The tech news sites I visit to learn about technological developments also feature heavy coverage of small startup companies. I sometimes read about scrappy two-person shops with no formal process, no required documentation format, and so on. I confess to occasional feelings of envy.

What can we do about these feelings?

We can shrug and get back to work. We can leave and join (or found) a startup. Or we can transform the job we have into the job we want.

How do we do this? There's no universal answer, and I'm certainly not presuming to give career advice to a room full of people more experienced than me. But I can at least tell you about what I'm doing. Maybe it's foolish. Maybe the other shoe will drop any day now and I'm going to have to stop doing this stuff. Until then, I'm going to keep using my favorite coping mechanism: experimentation.

Any given day presents us with little pockets of downtime: a meeting ends early, or a software build takes extra time. We can use those pockets to conduct experiments. We might download and try out a new programming language, set up a continuous integration server, or add a few tests to a subsystem that sorely needs it.

Or we might think a little bigger. We could learn our way around illustration software and use it to draw a user interface for an imaginary product. We could buy a cup of coffee for an industrial designer and ask about what's going on in their area of expertise.

Most of these little experiments won't result in anything tangible being made. Even so, they do us a service by challenging us. Every once in a while, an experiment might lead to a good idea. This is the part where we show it off to whatever receptive audience we can find: our teammates, our managers, etc.

Developing an idea and communicating it well enough to be adopted means we might get to spend time on it in a more official way, not just eking something out between compiles. In other words, we've just transformed a tiny piece of our career.

3. Conclusion

Over the past hour, we've discussed several different software practices. Some were code-level techniques, while others had more to do with project organization or personal development. None are things anyone would universally recommend.

So, why even bring them up? Because there are times when a blunt tool— a sledgehammer or a stick of dynamite—is exactly what you need to get a piece of recalcitrant software unblocked.

You can go your whole career making great software without doing any of the things we've discussed today. The real lesson here is: if you're stuck with a big puzzle in the form of legacy code, start somewhere. Start with something small you can do right now.

Don't get so paralyzed by analyzing new processes, new ways of working, or big changes that you end up not starting anything. What's the one thing you can do today that will take the most pain out of building software?

Delivering Quality in Software Performance and Scalability Testing

Khun Ban, Robert Scott, Kingsum Chow, and Huijun Yan
Software and Services Group, Intel Corporation
{khun.ban, robert.l.scott, kingsum.chow, huijun.yan}@intel.com

Abstract

We all are witnessing the amazing growth in today's computing industry. Servers are many times faster and smarter than those of just a few years ago. The need to ensure the applications run well with a large number of users on the Internet has continued and is getting more attention as the move to cloud computing is gaining momentum. To address the question whether the software system can perform under load, we look to performance and load testing. Performance testing is executed to determine how fast a system performs. It is typically done using a particular workload. It can also serve to verify other quality attributes, such as scalability, reliability and resource usage. Load testing is primarily concerned with the ability to operate under a high load such as large quantities of data or a large number of users. We use performance and load testing to evaluate the scalability of a software system.

In this paper, we characterize the challenges in conducting the application performance tests and describe a systematic approach to overcome them. We present a step-by-step data driven and systematic approach to identify performance scalability issues in testing and ensuring software quality through software performance and scalability testing. The approach includes how to analyze multiple pieces of data from multiple sources, and how to determine the configuration changes that are needed to successfully complete the stress tests for scalability. The reader will find it easy to apply our approach for immediate improvement in performance and scalability testing.

Biography

Khun Ban is a staff performance engineer working with the System Software Division of the Software and Services Group at Intel. He has over ten years of enterprise software development experience. He received his B.S. degree in Computer Science and Engineering from the University of Washington in 1995.

Robert Scott is a staff performance engineer working with the System Software Division of the Software and Services Group. Bob has over ten years of experience with Intel improving the performance of enterprise class applications. Prior to Intel, he enjoyed twenty years engaged in all aspects of software development, reliability and testing, and deployment.

Kingsum Chow is a principal engineer working with the System Software Division of the Intel Software and Services Group. He has been working for Intel since receiving his Ph.D. in Computer Science and Engineering from the University of Washington in 1996. He has published more than 40 technical papers and has been issued more than 10 patents.

Huijun Yan is a senior performance engineer working with the System Software Division of the Software Services Group. Huijun has over seven years of enterprise software performance experience. Huijun received her MS degree in Computer Science from Brigham Young University in 1991.

1 Introduction

Software performance testing has been a difficult challenge to master. It is as much an art as a science. To help meet the challenge, there have been great tool improvements in the recent years to help automate the testing processes and even generate test cases based on the application source code. There are also numerous commercial and open source tools to help with load generation by giving the user the ability to record interactions and replay them as if many users are performing those steps with some random think time between each step.

While the tools are improving, the problem set has also increased in size and scope. With the increasing computing capability of modern servers, it has become increasingly difficult to test the full scaling of the server as many external, often much slower, components can get in the way. Performance optimization has been a challenge for application servers [1]. In addition, modern platform features like power management can also complicate the testing by not giving us an accurate reading of the CPU activities. The principle goal of this paper is to outline the challenges we faced during performance testing and offer the solutions that worked well for us.

Enterprise software systems can involve many systems and components. Figure 1 illustrates one such setup. On the left, there are many clients interacting with the servers through some public interface such as web services, JavaServer Pages (JSP), Java Servlets, or just static HTML pages. Load generators are often used to simulate these users. The clients are then connected to multiple gateway switches that route the traffic to some faster inter-connecting switches which connect multiple database and application servers on the right. Additional storage devices may be attached as needed. In this paper, we concentrate on server software and restrict the system under test (SUT) to the servers and the components required to perform their function, i.e., the components inside the box in Figure 1. This does not imply that client system performance is not important, just that we can easily add more client capability when the client-tier becomes the bottleneck during testing. However, on the server we must identify any issue that may prevent it from scaling to support the maximum number of clients. In addition, clients may come and go, reboot whenever needed. Server systems are expected to have much longer up time when compared to client systems. As such, it is important to test the server to ensure that it can remain healthy and meet service quality levels for a longer duration and during peak usage. To meet this requirement, a load generator can be used to stress test the platform. Using the load generator, we are able to vary the duration and level of load on the system to understand how it performs in these different situations. In Figure 1, load generators would simulate many client users on the left side of the diagram. The right side of the diagram represents the SUT and will have some requirements to test for, such as supporting some number of users within some response time constraints.

It is not always possible to drive server systems to full capacity to meet the requirements due to different bottlenecks or issues in the setup. These issues can range from hardware constraints, operating system configuration, network settings, the choice of Java Virtual Machine (JVM) and its parameters, to multiple dependency locks. In most cases, these items do not become issues until the system is stressed under high load.

As the capability of the SUT grows, e.g., either through hardware or software upgrades, it may be necessary to augment the various components in the test setup. For example, if the current load driver cannot provide enough transaction loads to fully utilize the test system, a second driver system may need to be added. This process can domino to other components in the system. Additional loads processing may increase the demands placed against the database. If adding a second instance of the application on the SUT is required to fully utilize the platform's resources, this may require the addition of some form of load balancing. We can utilize each platform's performance metrics to monitor each resource and determine if there is a performance bottleneck. It is normal to need to go through this process multiple times during the life of an application.

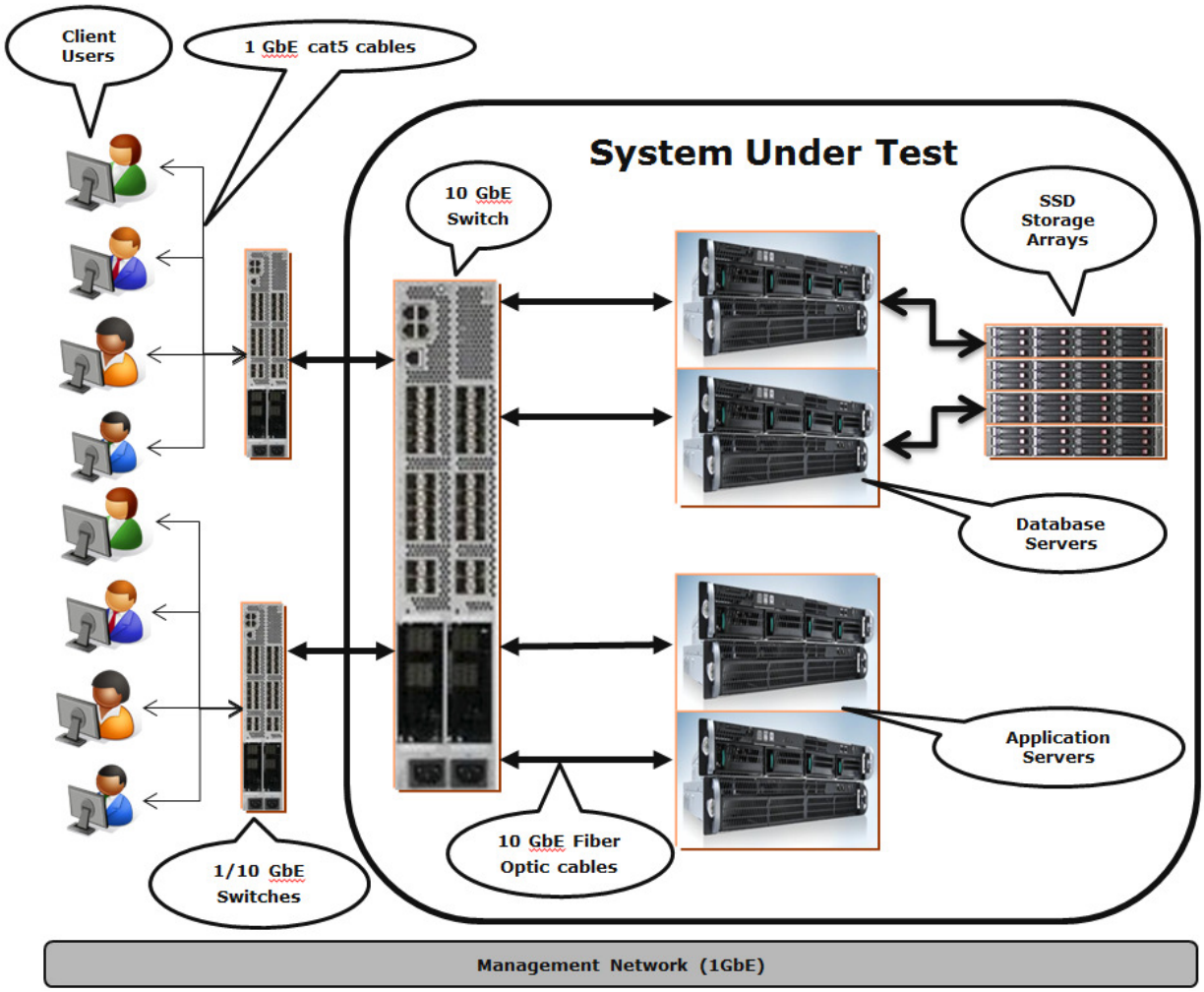


Figure 1. Performance and scalability testing for a multi-tier deployment environment.

Performance and scalability testing may run for hours, days, or even weeks. We need to consider memory and storage requirements. Will the memory management be able to keep performing well? What about the database system as more and more records are added? For example, in a short run of a Java application, the garbage collector may not need to collect the old space or perform a full collection. Instead, it may just work on the young space and leave the old space untouched to keep the garbage collection time short. For the database, accessing a small table with only few entries is usually fast. However, as the tables grow, indexes grow, and proper table design becomes important. Therefore, longer tests runs are required to detect these kinds of issues.

The contributions of this paper are:

1. Characterizing the challenges of performance and scalability testing, and
2. A systematic approach to overcome these challenges

2 A systematic approach to performance testing

Performance testing of enterprise software solutions usually involves multiple computers interconnected over a network. One popular deployment model is to separate the solution to multiple tiers. The mid-tier

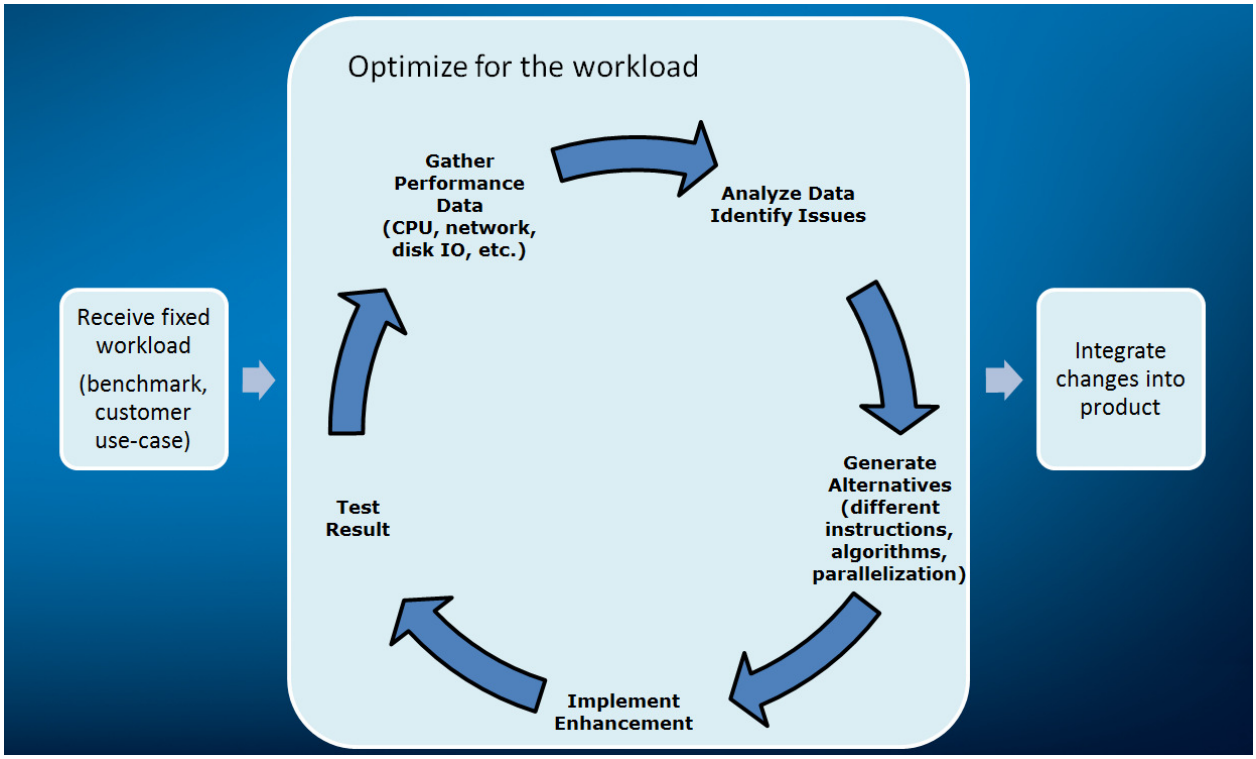


Figure 2. An iterative process for performance and scalability testing

generally handles the business intelligence and processing while the backend tier maintains the important information in some relational database. The client tier deals with the user interface and is not the focus of this paper.

Given the complexity involved, ensuring an adequate level of performance in this environment requires a systematic approach to avoid finger pointing, misdiagnosis, and focusing on the main bottleneck (root cause of the issue). Here we define a bottleneck as some resource that limits the performance of the solution as a whole. We assume that when the constraints on this resource are reduced, performance will increase.

We describe an iterative process to detect issues in performance and scalability testing in Figure 2. This process starts from the left when we are given a workload and some specifications such as the number of users to support and the response time required. From there, we will (1) gather performance data, (2) analyze the data to identify issues, (3) generate alternatives or enhancements that can be implemented, (4) implement enhancements and (5) test again to see if the fixes improve performance or meet the performance specification.

Once one issue is fixed, another issue may surface, so this approach will be repeated until sufficient bottlenecks have been resolved for the platform to perform to expectations. This does not mean that all bottlenecks have been exposed. A change to the software or to the hardware platform may cause new issues to surface.

Using the right tools is essential for measuring a platform's behaviors. On all Linux distributions, there are system performance counters, such as CPU usage, IO wait time, page swap activity, or context switches per second, that can be used to understand and diagnose performance issues.

SAR (System Activity Reporter, available on all Linux distributions) is a system monitoring tool that can take snapshots of the system performance counters. Obtaining periodic snapshots is an effective way of

finding performance bottlenecks and determining whether further action is needed. Several counters we found to be important to capture on all the system involved in the setup are:

- **%usr:** The percentage of time the CPU is spending on user processes, such as applications, shell scripts, or interacting with the user.
- **%sys:** The percentage of time the CPU is spending executing kernel tasks, such as file and network I/O.
- **%wio:** The percentage of time the CPU is waiting for input or output from a block device, such as a disk.
- **%idle:** The percentage of time where the CPU can perform additional work but no process is ready to run.
- **cswch/s:** This counter represents the number of context switches the system performs per second. A context switch occurs when the operating system scheduler switches execution from one process or thread to another.

In addition to knowing the average CPU usage, it is also important to check the utilization of the individual processors. For example, if a single processor has high utilization and other processors utilizations are low, the load being placed on the single processor may be a bottleneck.

If IO wait is more than 2-3%, we need to find the root cause and address the issue. Long IO wait time can be caused by either the system swapping if it is low on memory, or the system is performing large file IO operations. It is essential to know the root cause of the IO wait time. For Java, it is necessary to check the JVM parameters for heap size and utilization of large pages, and the system setting for the allocation of large pages. For example, if we have a system with 12 GB memory and we set aside 10 GB for 2MB large pages, we only have 2 GB left for Linux and other processes that must use small pages (2 KB pages). Then if we set the JVM heap to be anything larger than 2 GB and did not enable large pages for the JVM, the system will be swapping.

The number of context switches is another important metric to monitor for multi-threaded applications. If this number is high, it may be an indication that threads within the application may be blocked on some locks and unable to continue processing. What is considered high is dependent on the platform and the number of logical processors available. Knowing the value from a comparable application and platform can be used as an indicator.

Servers of today can dynamically adjust the operating frequency of their processors and memory, and even disable some processors when the load is light. This provides a huge savings of power, cooling, and cost to operate the system. The power saving features of modern processors may present a challenge to scalability and performance testing. For example, if the tools report 45% CPU utilization, this can be misleading if the cores are running at a reduced frequency of 1.6 GHz instead of the normal 2.93 GHz. For performance and scalability testing, it is sometimes helpful to disable power management and dynamic frequency adjustment to get the accurate reading on the system activities.

3 Case studies

Two case studies are presented in this paper to illustrate applying the iterative process for performance and scalability testing. In the first case study, we show that it is important to collect detailed SAR performance data to root cause the problem. In the second case study, we show that additional data beyond SAR system performance data are needed to isolate a hardware configuration change.

3.1 Case Study 1

This scenario describes how the average CPU usage on the system can be a misleading indicator. In this case, the workload measures the amount of HTTP traffic between two systems. The test environment consists of 2 systems; a driver and an application server. The driver machine uses Faban [2] to send requests to the AppServer, and the AppServer responds to the driver's request. The performance metric is operations per second (ops); the higher the better. A customer reported that running this workload on

the newer server with more cores resulted in 10% lower performance than on their older system with fewer cores. The older system had two quad-core CPU running at 2.67 GHz with 4 MB of L3 cache. The newer system had 2-socket 6-core CPU running at 2.33 GHz with 8 MB L3 cache. On the new system, the average total CPU usage (user + system, no IO Wait) was only 27%. This workload was not very CPU demanding so they did not understand why they were observing lower performance.

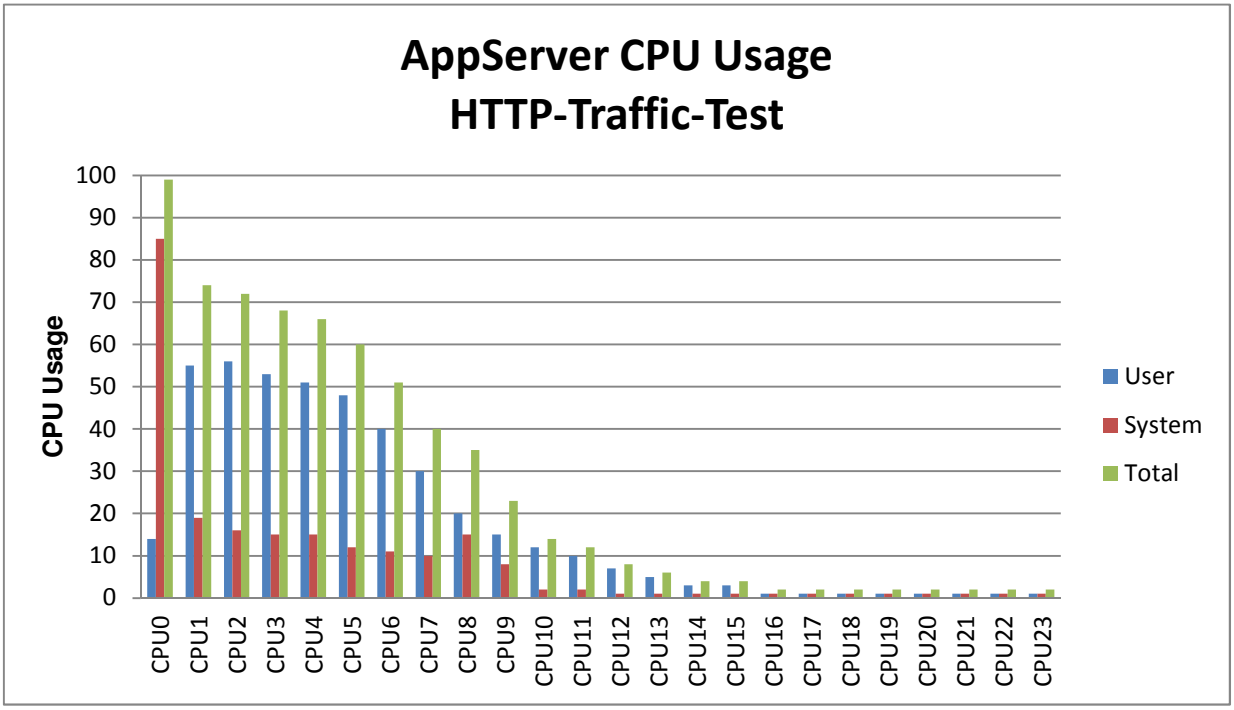


Figure 3. Case Study 1 – Distribution of utilizations across the CPUs.

In order to diagnose and solve this low performance issue, we performed the following steps.

1. We collected performance data from both the older and the newer system. We confirmed that the older system was performing better.
2. Using SAR performance counters data collected during step 1, we observed the CPU distribution outlined in Figure 3. It is quite clear the CPU 0 of the new server was heavily used even though the overall average was less than 30%. It turned out that the system interrupts were mainly handled by CPU 0 after examining the interrupts distribution (/proc/interrupts). The behavior was similar for the older system. However because the older system was running at faster frequency with fewer cores, the effects of this were less severe. We recommended to our customer to balance interrupt handling.
3. We modified the system interrupt handling to distribute the interrupt handling across all the cores equally. We then reran the workload and recollected the data.
4. The performance increased on both the older and the newer system. Additionally, the newer system now performed 20% better than the older system.
5. Then the customer asked why we cannot get even higher performance on the newer server since it still has more CPU headroom.
6. Beside CPU usage, SAR also reported network utilization. On our setup, we have only one 10 Gb/s connection between the driver and the application server. SAR reported our network usage was 9.5 Gb/s. Therefore the network was saturated and had become the new bottleneck.
7. The workload is no longer CPU bound; the available CPU power cannot push performance higher without first adding additional network capability.

3.2 Case Study 2

Sometimes even a good feature like power management can have unintended effects. A platform can reduce the frequency of its processors and save power if the processor's load is low (even for a few milliseconds). Today's platforms can also increase a processor's frequency if the surrounding cores are idle; keeping the same power level but increasing single threaded performance. One customer reported that their single threaded application was running much slower than expected after a BIOS upgrade to the application server. Their application consumed about the same CPU as before the upgrade, 10% total CPU usage, but achieved 50% less throughput and had a 50% higher response time.

Prior to upgrade, all power management features had been disabled so the processors would run at the default frequency (3.33 GHz in their case) all the time. After the upgrade, power management was enabled by default. Because the application required very little CPU, the power management feature reduced the processor frequency to just 1.59 GHz from 3.33 GHz. Therefore even when the CPU usage reported by the OS was about the same, the application was not consuming the same amount of CPU cycles as at the higher frequency. This issue was fixed by disabling power management features on the platform. Note: power management features will have little performance impact if the platform is typically driven at higher load levels.

4 Summary

Software performance testing involves testing not only the software application but also the underlying hardware, its configuration and usage. A failure to reach the specified performance criteria can come from not only the software application itself, but also from the server platform, and/or from the network and system configurations on all systems involved.

When running high loads in conducting scalability and performance testing, it is important to collect adequate sets of system performance metrics to ensure the performance criteria is met, and help to root cause when it is not. We have provided an approach to address the problem. The approach includes a set of system performance counters to collect and how to interpret the data collected. By applying the analysis from the system performance counters to fix the problems in the test environment, we can increase the confidence of the performance test results. We presented case studies to illustrate the problem if the performance test methodology is incorrect, a wrong conclusion may be drawn from an otherwise correct piece of software application that has been written to the specification. We also shared a set of best practices for performance and scalability testing.

Acknowledgments

Jonathan Morris reviewed and provided helpful suggestions to improve the clarity of the paper.

References

1. Khun Ban, Kingsum Chow, Yong-Fong Lee, Robert Butera, Staffan Friberg, and Evan Peers, "Java Application Server Optimization for Multi-core Systems" http://software.intel.com/en-us/articles/java-application-server-optimization-for-multi-core-systems/
2. Faban Harness and Benchmark Framework. http://java.net/projects/faban/

Perf Cells: A Case Study in Achieving Clean Performance Test Results

Vivek Venkatachalam (vivekven@microsoft.com)
Shirley Tan (shirleyt@microsoft.com)
Marcelo Nery dos Santos (marsant@microsoft.com)

Abstract

A key indicator of the quality of a software product is its responsiveness and performance, as this enables users to get tasks done quickly and efficiently. Test teams realize this and therefore typically set aside time and resources for performance testing. Unfortunately, the effort inevitably runs into the dreaded "high unexplainable variability in performance test results". Since fixing this is hard to do, a typical workaround is to raise the acceptable loss threshold to ensure only large performance trigger corrective action. However, this causes a "death-by-a-thousand-cuts" effect where numerous small performance losses (that are all real) are allowed into the product. By the time the product is ready to ship, these small performance losses have accumulated and the product exhibits poor performance with no easy fix in sight.

How does one get close to the ideal of a performance test system that can reliably detect even small performance losses on a build-over-build basis while minimizing false positives? This paper describes our attempt to tackle this problem while running performance tests for Microsoft Lync.

The bulk of the variation in performance results when testing a distributed system is typically due to network traffic variations and varying load on the servers. Our approach to eliminate this variability was to design a system where the performance test runs on a virtualized distributed system, that we call a Perf Cell. This is essentially a single physical machine that houses all the individual components on separate virtual machines connected via a virtual network and otherwise isolated from all external networks. In the paper, we present our design and implementation as well as results that indicate that variability is indeed reduced. We believe this paper would be useful reading for engineers responsible for designing, implementing or running a performance engineering system.

Biography

Vivek Venkatachalam is a software test lead on the Microsoft Lync team. He joined Microsoft in 2003 and worked on the Messenger Server test team before moving to the Lync client team in 2007. He is passionate about working on innovative techniques to tackle software testing problems.

Marcelo Nery dos Santos is a software engineer on the Microsoft Lync test team. He has primarily worked on tools to help test performance for the Lync product. His main interests are working on innovative tools and approaches for enabling more efficient testing.

Shirley Tan is a software engineer on the Microsoft Lync test team and recently completed 5 years at Microsoft. Apart from her work on performance testing, she is interested in Model Based Testing and Code Churn Analysis.

1. Introduction

As competing software products grow increasingly similar in terms of functionality and feature set, aspects such as performance, security, accessibility etc. are what set them apart. Performance is arguably among the more important of these aspects. Given two software products that provide a comparable set of features, customers will tend to prefer the product that allows them to accomplish their tasks in the quickest and most efficient manner. This implies that software teams should pay careful attention to the performance of their products to maximize customer satisfaction. The best way to do this is to incorporate a formal performance engineering process into the product development life cycle and to keep making improvements continually to both the process and the product.

A performance engineering system, in our opinion, is not really very different from any other engineering quality control system. At its core, a well-designed performance engineering system allows the product team to a) prevent unintended performance losses from creeping into the product by monitoring daily builds for degradations in performance and b) reliably verify that any fixes made actually fix the problem. A key obstacle one faces when designing/implementing/running this system is that the performance of the product under test could exhibit variations due to factors external to the product (e.g. state of the system under test, state of the network etc.). We propose the term “noise” to collectively refer to all such factors because they mask the “signal” that we care about (performance changes directly attributable to specific changes in the product code). Noise in the engineering system is a big problem because by masking the signal it becomes difficult to distinguish real performance problems in the product code from artificial/transient problems caused by these external factors. As members of the Microsoft Lync client performance test team, the authors directly encountered this problem during the Lync 2010 ship cycle and this paper is essentially a firsthand account of our experience with this problem and our attempt to solve it using virtual machines.

We start off by giving the reader a brief background on the Lync product and an overview of the performance engineering process we followed during the Lync development life cycle. Section 2 also serves as a high-level introduction on the key things to consider while setting up a performance engineering process for the benefit of readers unfamiliar with this domain. In section 3, we delve into the problems we faced because of the unexplained variations in our performance results and the repercussions of those problems. In sections 4 and 5, we then describe our solution and present details on the design and implementation of our solution to minimize noise in the performance engineering system by using virtual machines. Finally in sections 5 and 6, we present our results from this solution, key takeaways and future improvements.

2. Lync and the performance engineering process

This section will describe the multiple phases of our performance process and bring some considerations on the initial performance tests that were executed.

2.1. Lync – a brief overview

Microsoft Lync[1] is an enterprise communication and collaboration system that offers users a rich set of features (instant messaging, user presence, voice/video calls and conferencing, content sharing among many other features). It comprises of a set of server components (collectively known as the Lync server topology) and a set of clients that run on multiple platforms (Windows, Mac, Web client amongst others) and connect to and communicate with the server components to provide this rich functionality. The authors were on the team responsible for verifying the performance aspects of the flagship Lync client,

[1] http://lync.microsoft.com/

the client running on the Windows platform and all future references to Lync in this paper imply this specific Lync client.

As should be clear from this description, the Lync system is a distributed system and the majority of the functionality of the Lync client involves communication with the server components and through them, communication with other Lync clients if needed. This causes some unique issues while attempting to test the performance characteristics of any one instance of the Lync client, which is discussed later in the paper. With this basic understanding of the Lync product in place, we now give the reader a basic overview of the performance engineering process we followed.

2.2. Performance Engineering Process Phases

2.2.1. Planning

During the planning phase, we identified the set of key scenarios that we wanted to focus on improving, as well as the key metric we wanted to track. There are many performance related metrics one can track (elapsed time, CPU usage, disk usage, network usage, memory usage, power usage etc.) but in the interests of optimizing the cost/benefit ratio of our effort, we decided at the very outset to focus on the elapsed time metric, since that made the most sense for a real-time application such as Lync.

Picking the elapsed time metric does not mean that we ignored all other metrics. It is just that our performance exit criteria for the release were based on whether or not we met our elapsed time goals for the set of scenarios we had identified. The performance scenarios and goals themselves were defined based on feedback from customers, the real world user environment and extensive meetings with the members of the product team that owned designing and implementing those specific scenarios.

Some examples of scenarios and associated metrics that we decided to target are:

- Time it takes for a user to be signed in from the time the user clicks the “Sign In” button.
- Time it takes for a call to be connected from the time the user accepts an incoming call.
- Time it takes for a desktop sharing session to be setup from the time that the user clicks the “Start Sharing” button.

In addition to defining the scenarios and associated goals, we also defined a range of user models which covered things like what the hardware specifications should be for client machine used to run these tests, what software environment these machines had (operating system version, 32-bit vs. 64-bit etc.), and external considerations such as the kind of network connection (corporate intranet, broadband DSL/Cable, dial-up modem) etc. Once again, to optimize the cost benefit ratio of our effort, we decided to focus all our tests on a very specific user model (user on specific kind of laptop, connected to the corporate intranet on high-speed Ethernet etc.). This also served the purpose of defining a consistent environment in which we would run our tests and eliminate variability in performance results due to these factors. We still ran performance tests for other user models as part of sanity testing but the majority of our tests were run using the identified target user model.

2.2.2. Product Source Code Instrumentation

Once the scenarios were defined and appropriate goals set for them, the product source code was instrumented using the Event Tracing for Windows (ETW) Toolkit[2]. The APIs were used to insert ETW

[2] A system that provides application programmers the ability to start and stop event tracing sessions, instrument an application to provide trace events, and consume trace events. http://msdn.microsoft.com/en-us/library/bb968803(v=VS.85).aspx

events in appropriate locations for each of the target scenarios. For example, instrumenting the SignIn scenario meant inserting code (using the ETW API) to:

- Fire an ETW event called SignInStarted in the very beginning of the method that handled the SignIn button click.

- Fire an ETW event called SignInCompleted at the very end of the method that updated the UI to indicate to the user that she was now signed in.

When this code then executes, the ETW system fires an event that contains in its payload, a very precise timestamp for when the code was executed. The event payloads can also include state information specific to the event. This user-level payload is optional but can be used to uniquely identify events based on this data. The ETW system is highly optimized and therefore the instrumentation itself adds no overhead to the performance of the product. In fact, if there are no active ETW consumers listening for these events (the normal situation on a customer's machine) then the ETW system does nothing and no event is fired.

2.2.3. Test Design and Execution

This involved driving one or more Lync clients to exercise the performance scenario. We started by running these tests manually while building up the capability to run these tests in an automated fashion. First, we needed to create a mechanism to automate UI actions on an individual Lync client. Once we had built this, we then needed to design a distributed test harness that could enable us to run UI actions on multiple Lync clients in a synchronized manner. We next had to make a decision on the number of times to run each scenario. Ideally we would have liked to run each scenario 20 to 30 times to make sure summary statistics such as averages and standard deviations were meaningful. On the other hand we had to consider the fact that we needed the entire set of tests to run within a reasonable period of time. After careful deliberation and some experimentation we decided to run each scenario 5 times. Each time a scenario was run, the associated ETW events would fire because the performance test harness explicitly setup an ETW consumer to listen for these events. For example, the UI automation for the SignIn scenario would click the SignIn button and wait for the client to be signed in, which in turn would cause the associated ETW events to be fired and saved in a log file. At this point the log file essentially contained a list of entries, each of which contained information about an event such as the timestamp at which these events were fired. In addition to the ETW log files, we also collected the product log files to aid in investigation of any problems that came up.

2.2.4. Log processing

At this stage, all scenarios had been run and relevant log files been collected. This involved parsing each of the ETW log files generated in the above step matching start and stop events for each scenario, and calculating elapsed time by computing the difference between the timestamps of the stop and start events and then dumped into a database.

2.2.5. Analysis and Reporting

Summary views of this data were then exposed through a Silverlight[3] based interactive reporting portal so that stakeholders could easily view this information and stay informed on the overall status of the Lync client as it pertained to performance scenarios and associated goals. The engineering team would use this reporting portal to quickly identify scenarios whose average elapsed times were above stated goals for the latest set of test runs and start a thorough investigation into the problem. This investigation involved collecting and poring through associated log files from all the components to try and understand

[3] Silverlight is a Microsoft development platform for creating engaging user experiences. http://www.silverlight.net/

where the delays appeared to be occurring and why. Once we had narrowed down the cause via investigation of the logs, the normal test process would then be followed. The test team would file a bug to track the issue, assign to the appropriate developer, who would then design and implement a fix, and assign it back to the tester. Finally the tester would verify that the problem had been fixed in subsequent test runs and close the associated bug.

3. Overview of the problem

Since our primary goal was to track the overall response time faced by the user in a realistic environment, we had made the choice early on to run these tests against a topology that was actively used by members of the Lync team for their day-to-day work. Prior to release of the product externally, this was the best way for us to get performance data in a realistic setting. These tests were run approximately once a week on the latest build and the data from these tests was used to get a sense of how that build fared with respect to meeting the goals set for each scenario. As described earlier, each week we would go through the report from the most recent test run and identify scenarios that were not meeting the target goals and start investigations into each one of these.

Over time, we noticed that a large percentage of these investigations led to dead ends because there was nothing obvious about what the cause was. To make things worse, every once in a while, a scenario that was previously above its goal would automatically get back within its goal in subsequent test runs without any relevant code changes having been made in the interim. Gradually, it became clear that there were unexplained variations in the elapsed time results for a given scenario across test runs and this was making it difficult to distinguish real performance degradation due to code changes introduced in a new build from random variation in the results. While the causes of these were unknown, it was clear that we could no longer use our process to detect small performance degradations reliably. We were forced to ignore any small performance losses that the data was indicating because it took a non-trivial amount of developer/test engineering effort to investigate every such issue and frequently these investigations could not pinpoint any specific cause. In each such case, we were then left with no choice but to conclude that the performance degradations were transient issues caused by noise in the system and not due to a real issue in the product. Essentially, we were spending critical resources in wild goose chases while at the same time management was slowly starting to lose trust in the ability of our performance engineering system to accurately evaluate performance of the Lync client and to find real issues in the product.

Due to the time and resource crunch during the ship cycle, we could not afford to spin up a separate process to get to the bottom of the issue. So we chose the next best solution, which was to make the decision to consciously focus only on obviously large degradations that stood out. The good news was we managed to ship a high quality Lync with this process but we knew we needed a better mechanism going forward.

4. Our Solution- The Perf Cell

After Lync was released to manufacturing, we conducted many post-mortem and brainstorming sessions to discuss how to improve the overall performance test process with special emphasis on identifying and eliminating sources of variability in our engineering system. Over these sessions, one key issue soon became clear. We were attempting to achieve inherently incongruent goals by running a single set of tests to determine both that:

a) There were no performance degradations since the last tested build;
b) The product was meeting goals against real world topologies.

We had attempted this kill-two-birds-with-one-stone strategy by running all our tests against a real-world topology which was a reasonable solution for the latter but a terrible solution for the former because of the inherent variability in a real-world system.

We also realized that detecting and fixing performance degradations as soon as they were introduced was the best way to ensure that the product would meet its goals at the end of the product cycle. This was crucial since performance is lost not by a small number of large losses (which are usually easy to detect and fix) but by a large number of small losses that build up over time (which are usually difficult to detect and also harder to fix). Once we had internalized this, a couple more brainstorm sessions helped us arrive at the innovative concept of what we call a Perf Cell.

A Perf Cell is essentially a self-contained virtual test topology that contains all the necessary server and client components, all running as virtual machines (using Hyper-V[4]) on a single physical machine.

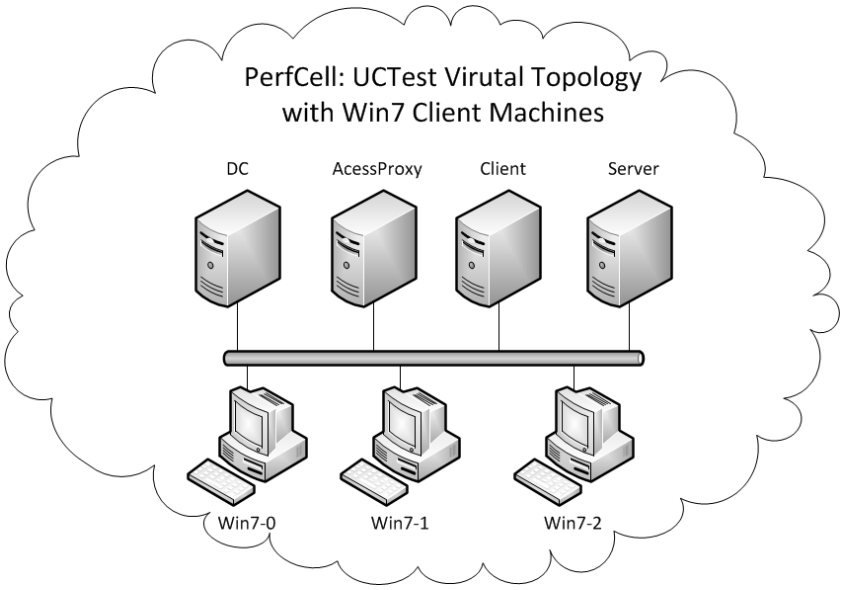


Figure 1: Perf Cell Virtual Topology Diagram

Given a physical machine that meets the necessary minimum requirements (Memory >=12GB), the Perf Cell topology is generated automatically by a script that sets up the whole system starting with bare metal. The test network is completely isolated from other networks and is completely self-contained since all the necessary server components are available on it. On average, with this script in hand the deployment time for building a fresh Perf Cell from scratch was about 4 hours and once we had the basic Perf Cell deployed, updating existing components in place was much faster. So overall, we were already seeing advantages in terms of the time and resources it took to setup an entire performance test lab.

This design clearly provides the following key benefits:

- **Eliminate extraneous network traffic**: As noted before, all clients and server components in the Perf Cell were connected via a virtual network which was disconnected from the rest of the network. This avoids traffic on external network resources and helps minimizing network variability.
- **Ensure consistent load on the server**: This was automatic because the only load on these server components was from the test clients we were actually running.
- **Easy management of the test system:** Instead of having to maintain a network of individual machines and the network components, we really only needed a single physical machine with appropriate hardware specs and our script which could be completely automated.

[4] Hyper-V is the virtualization technology provided by Microsoft. http://www.microsoft.com/hyper-v-server/en/us/default.aspx

- **Cheap and fast deployment:** With the deployment scripts mentioned before, we could roll out a self-contained performance test environment in less than half a day using automated scripts and for the cost of one server class physical machine. This is cheaper when compared to the process of setting up a more realistic topology with many physical machines, both in terms of cost of machines and then time to deploy them. In addition to the physical machines, there are also networking issues one has to deal with to get this setup running. These benefits are possible thanks to the magic of virtualization and Hyper-V.
- **Scale Out:** This is a corollary of the above benefit. Since the only constraint is the availability of physical machines, the tests can be partitioned and run in parallel (limited only by budget available for paying for additional host machines[5]) to speed up overall execution time. In practice this meant that instead of running tests once a week against a build from a specific branch, we could run these tests daily if we wanted against a build from each branch.

5. Design of the Data Collection/Analysis/Reporting system

Once we had designed the Perf Cell, we designed and implemented a lightweight performance test harness that we could easily use to run our multi-client UI automation tests on the virtual client machines in the Perf Cell. The harness is based on Windows Communication Foundation (WCF)[6] and allows the multiple virtual machines to be remotely controlled by a process that runs on the physical machine that hosts these virtual machines. The controller process on the host machine runs a set of scripts that co-ordinate tests across all the virtual client machines hosted on that machine (win7-0, win7-1 and win7-2 in Fig 1 above). It is worth noting that a single real machine exists and besides hosting the virtual machines, it performs a double duty by also executing the controller process.

For a given scenario that needs to be exercised, the controller process launches the corresponding set of tests on each virtual machine. Each test then executes independently on its virtual machine and drives the Lync UI on that machine as needed. Synchronization between tests running on independent machines is achieved by a lightweight synchronization mechanism based on a Publish-Subscribe (1) model. Test cases on individual virtual machines are able to post messages with the controller process running on the host machine indicating that they are done executing some significant part of the test. Similarly they can indicate to the controller process that they are waiting for messages from other machines and block test execution until the controller passes on the expected message from the expected source endpoint. This mechanism ensures that every test process is autonomous and could work independently until the point where it needs to sync with the controller or with other test processes. This approach is easy to manage and the main controller only needs to start the broader test scenario and does not need to control every single step of test execution. A brief overview of the system is provided below:

1. The controller script running on the host machine reads a configuration file and launches a test case process on each client virtual machine. It passes some parameters so the actual test case binary knows which role it is playing, which test account to use, the logging directory to be used and other parameters. The controller script is also parameterized, so it is flexible enough to execute just a subset of test cases.
2. The test case process is responsible for making sure that appropriate performance counters are collected during the test execution. The system is flexible enough that any sort of data may be

[5] A host machine is the physical computer that is running one or more instances of a virtual machine.

[6] WCF is a set of Microsoft technologies that enable development of distributed systems. http://msdn.microsoft.com/en-us/netframework/aa663324

collected, such as ETW traces, system information (like CPU, memory, IO, Registry operations, etc.) or any other custom performance indicator the product may have. During the test execution, log files are created and performance results are written to the log location.

3. A completely separate log analysis process is responsible for analyzing the performance data. This approach fully decouples the test execution from the performance analysis and reporting. It also allows a simple extensible model where a handler on the analysis side can be easily developed to parse the results for any new set of performance data collected by the tests. As an example, we may have an analyzer which deals with ETW traces, another one dealing with memory dumps or an analyzer which parses custom data the test wrote during test execution.
4. The performance infrastructure is structured in a way that multiple test machines can run in parallel, writing their test results to a log store. The log analysis process checks for new data in the log store in real time and analyses new data as soon as it is available. The design of the system allows for multiple log processors to run in parallel and thus enable quicker analysis of test results.
5. The end results generated by the analyzers are stored in a database and can be accessed through a rich reporting portal built using Silverlight technology which minimizes the time it takes for new visualizations to be shown to the user. Once a set of data is retrieved, the user is able to fully interact with it until new data needs to be retrieved from the back end. The performance of the backend database was also highly considered and we currently have over a million data points being served with high performance, where we are able to filter data and show results in a few seconds.

6. Initial Results

Our preliminary results from analyzing performance data for a test run against a Perf Cell are very promising. The variation in the results appears to be drastically lower compared to variation in the results from a run against a real-world topology.

The following two charts illustrate the difference between elapsed times for scenario A when tested against a real topology and a virtual topology. The charts are essentially histograms showing the frequency distribution of elapsed time measurements for this scenario. The elapsed times for each time are slotted into a bin and the height of the bin indicates the number of elapsed times that fall in that bin (indicated by the label Measurement Count on the vertical axis) and the x-axis simply denotes the entire range of elapsed times that were noticed).

Figure 2 shows elapsed time data for tests run against a real topology and one can see that the distribution appears to be approximately log-normal (2) with a long tail (the overall range of times is broad and there is no clear clustering of data, making it hard to judge the performance for this scenario). Since there is a risk that the logs might show no clear cause for why certain test iterations show higher elapsed times, we would have to assume that this was just caused by noise in the system and skip an investigation altogether in the interests of saving engineering resources. By making this choice, we automatically run the risk of letting some real degradation get into the system. The only situation in which we would notice a real degradation is when the entire histogram was shifted to the right and there was obviously something very wrong with the performance of the scenario. That is when we would decide it was worth spending the time to investigate the problem because of the high probability of finding a real issue.

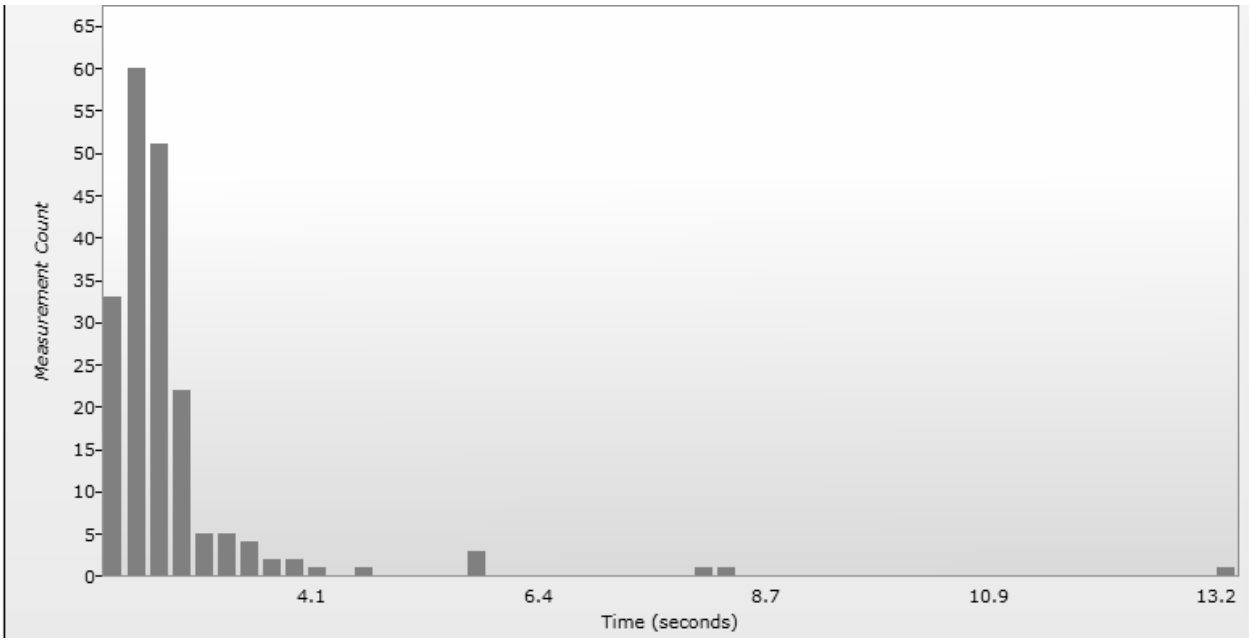


Figure 2: Histogram showing frequency distribution of scenario A's elapsed time results from performance tests run against a real topology

In contrast, in Figure 3, where a Perf Cell was used, one can see two tight clusters. The first cluster contains more than 90% of the data in the 0.695 to 0.997 range which indicates very low variability in the results, thus validating the overall design of the Perf Cell concept. We do see another cluster (approximately 6%) in the 15.57 to 15.87 range. This is a classic error condition where a given operation succeeds only after one or more retries and therefore the elapsed times are much longer. The fact that even the outliers are tightly clustered is further evidence that use of the Perf Cell is successfully eliminating most of the inherent variability in the system and increasing the power of the tests to notice even small degradations in the performance of the software. In such a system, we no longer have to make a choice to not investigate small degradations. Due to the low level of noise, we can be reasonably confident that even small degradations indicated by the data are real issues and it is worth our time getting to the bottom of the issue. This in turn means that there is a much lower chance of letting real performance degradations fall through the cracks and get into the product.

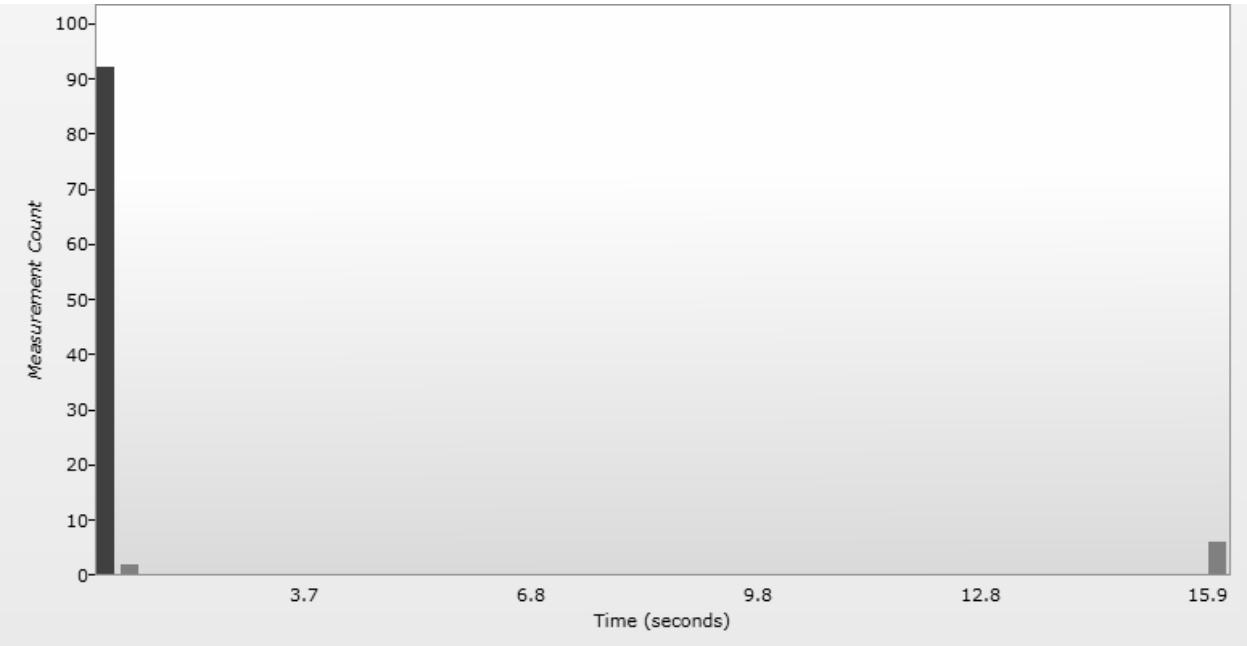


Figure 3: Histogram showing frequency distribution of scenario A's elapsed time results from performance tests run against a Perf Cell

As we can see from the above figures, the elapsed times that we observed from tests run against the Perf Cell exhibit much less variation enabling us to more clearly see real aberrations that merit further investigation. This helps us to increase the sensitivity of our tests in catching even relatively small performance degradations and help keep the product quality high throughout the cycle.

7. Conclusion

Hopefully the reader now has a general understanding of the things to consider when designing a performance engineering system. In our opinion, the most important takeaway is that the system needs to be engineered very carefully to eliminate as many sources of noise i.e. test interference as possible. As always, this needs to be balanced by the needs of the team and the resources available to contribute to this system.

Based on the results we have seen thus far, we believe the Perf Cell concept is perfect for running performance tests that are sensitive enough to catch even relatively small degradations in performance across different builds of a software product. We believe that using a Perf-cell based design is especially effective for testing distributed systems because:

- It eliminates extraneous network traffic from the system (all traffic between client components and server components is on the virtual network which is isolated from the rest of the world) which is frequently a cause for transient performance degradations that are not associated with any changes in the product.

- It greatly reduces (if not outright eliminates) variability caused by varying response times from the server during the test since the only traffic to the server components is the traffic sent by the clients actually in the test and the server is always in a known steady state.

- It allows for a relative efficient way to scale out the test effort by partitioning the tests into natural groups (for example, voice call tests, video call tests, instant messaging tests etc.) that can all be run in parallel on their own Perf Cells. With the increasing availability of computing resources in

the cloud, this easily lends itself to the possibility of spinning up Perf Cells in the cloud on demand for the duration of a test run and tearing them down after the run.

- It allows for relatively easy modifications to yield more improvements. One improvement that we are considering is the ability to test two adjacent builds simultaneously instead of over separate days. Instead of running tests against say, build A on Mon and build A+1 on Tue, we could increase the number of clients available in the perf cell (by simply modifying the script appropriately) and test build A and build A+1 simultaneously. This will allow us to further reduce noise by elimination noise caused by temporal factors (e.g. the state of the system on Mon at the time of the test is likely to be slightly different from the state of the system on Tue and by running tests against the builds from Mon and Tue simultaneously on Tue, we eliminate variability in results caused by this difference). By doing this, we expect to eliminate temporal variability and further increase the power of our performance tests to detect the smallest of performance degradations.

References

1. Publish/subscribe. *Wikipedia.* [Online] http://en.wikipedia.org/wiki/Publish/subscribe.

2. Lognormal Distribution . *NIST/SEMATECH e-Handbook of Statistical Methods.* [Online] http://www.itl.nist.gov/div898/handbook/eda/section3/eda3669.htm.