TWENTY- EIGHTH ANNUAL
PACIFIC NORTHWEST
SOFTWARE QUALITY CONFERENCE

October 18-20, 2010

World Trade Center Portland
Portland, Oregon

PNSQC 2010 President's Welcome

Software engineering has grown significantly on both sides of "the fence". There are more methods, techniques, tools, and know-how; yet there is also more demand for better and more useful functionality that lasts longer, breaks less, and works more easily across a wider array of platforms. And the span of "cross-functionality" is ever-widening as organizations grow more complex, and more job functions, geographies, and application areas are involved.

So how are we to achieve, and even improve, quality in this evermore complex environment?

Welcome to the 28th annual Pacific Northwest Software Quality Conference! This year we had more paper submissions than ever before, and we've greatly expanded the number of poster papers.

We are excited to have Tim Lister kick off the conference with an exploration of how project patterns can be more powerful than even "best practices" for guiding software projects. As co-author of Peopleware and other classic books on productivity and risk management, and with over 35 years of professional software development experience, Tim is truly one of the masters who can provide valuable insights and breakthrough ideas.

Our second keynote, Harry Robinson, has rich software development and testing experience at AT&T Bell Labs, Hewlett-Packard, Microsoft, and Google that has helped him understand both the big picture and the trenches of software testing like only a special few. Harry will describe the Bing team's journey over the last year at Microsoft to simplify and improve their testing, and in doing so will share several gems of learning.

We have modified our traditional birds of a feather lunch discussions this year to now be larger, expert-led, interactive discussions. The Tuesday lunch will have lightning talks. Additionally, we will have ~30 poster papers Monday evening and Tuesday, and there will be a Rose City SPIN presentation Tuesday evening. Look for descriptions and how to participate in these informal, interactive activities during my conference opening remarks. Also, descriptions of these activities will be broadcast on Twitter. And we hope you check out our new online forum on www.pnsqc.org.

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 2010

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 18

Project Patterns: From Adrenaline Junkies to Template Zombies **1**
Tim Lister

Keynote Address – October 19

Using Simple Automation to Test Complex Software **9**
Harry Robinson

Test Technique Track – October 18

Contextually-driven System Architecture Reviews **15**
F. Michael Dedolph

Effective Testing Techniques for Untold Stories in Story-Driven Development **31**
Erbil Yilmaz and Gokhan Ozer

Simulating Real-World Load Patterns When Playback Just Won't Cut It **39**
Wayne Roseberry

Testing as a Risk Management Activity **55**
Alan S. Koch

Automation Track – October 18

Test Environment Configuration in a Complex World **63**
Maxim Markin, Edward French and Liu Hong

Issues in Verifying Reliability and Integrity of Data Intensive Web Services **73**
Anand Chakravarty

Peering into the White Box: A Testers Approach to Code Reviews **81**
Alan Page

Testing Concurrency Runtime via a Stochastic Stress Framework **89**
Atilla Gunal and Rahul Patil

Using Silverlight PivotViewer to Make Sense of the Chaos **107**
Max Slade, Frederick Fourie, Melinda Minch and Ritchie Hughes

Streamlining Test Automation through White-Box Testing Driven Approach **123**
Sushil Karwa and Sasmita Panda

Process Improvement Track – October 18

Quality Pedigree Programs: Or How to Mitigate Risks and Cover Your Assets **133**
Susan Courtney, Barbara Frederiksen-Cross, and Marc Visnick

Inspiring, Enabling and Driving Quality Improvement **145**
Jim Sartain

Document Your Software Project **153**
Ian Dees

The Last 9% Uncovered Blocks of Code – A Case Study **163**
Cristina Manu, Pooja Nagpal, Donny Amalo and Roy Tan

Managing Polarities in Pursuit of Quality **175**
Denise Holmes

Lean System Integration at Hewlett-Packard **187**
Kathy Iberle

Collaborative Quality and Performance Track – October 18

Bridging the Cultural Gap **205**
Katherine Alexander

Performance Testing and Improvements Using Six Sigma – Five Steps for Faster Pages on Web and Mobile **213**
Mukesh Jain

Testing the Mobile Application's Performance: Case Study on Windows Mobile Devices **227**
Rama Krishna Pagadala

Increase QA Team Efficiency by Utilizing Offshore Resources **241**
Hao Zhao

Lessons Learned About Distributed Software Team Collaboration **245**
Kal Toth and Raleigh Ledet

Agile Track – October 19

Scaling Agile Practices; Replicating the Small Team Achievements to Larger Projects **257**
Don Hanson

Acceptance Tests Driving Development in Scrum **263**
Sari Kroizer

Person-to-Person Communications: Models and Applications **277**
Rick Brenner

The Good, Bad, and the Puzzling: The Agile Experience at Five Companies **287**
Michael Mah

Testing and Testing Technique Track – October 19

Don't Test Too Much! *[or too little] (Lessons learned the hard way)* **295**
Keith Stobie

Software Quality Assurance in the Physical World **303**
Kingsum Chow, Ida Chow, Vicki Niu, Ethan Takla and Danny Brillhart

Visualizing Software Quality **313**
Mark Fink

Using Static Code Analysis to Find Bugs Before They Become Failures **323**
Brian Walker

Adaptive Application Security Testing Model (AASTM) **333**
Ashish Khandelwal and Gunankar Tyagi

Large-Scale Integration Testing at Microsoft **347**
Jean Hartmann

Usability, UI and Complex Data Track – October 19

User Experience Grading via Kano Categories **357**
Matt C. Primrose

Affecting Printer Installation Success in the Consumer Market **363**
Kathleen Naughton

Customer-Driven Quality **373**
John Ruberto

Incorporating User Scenarios in Test Design **389**
April Ritscher

Driving Product Quality towards Release Goals **397**
Bhushan Gupta

Working with Complex Applications **407**
Engin Uzuncaova

Complexity, Automation and Process Improvement Track – October 19

Turning Complexity into Simplicity **417**
Jon Bach

Coding vs. Scripting: What Is a Test Case? **427**
Sam Bedekar and Julio Lins

Web Test Automation Framework with Open Source Tools Powered by Google WebDriver **445**
Kapil Bhalla and Nikhil Bhandari

Assessing the Health of Your QA Organization **455**
Michael Hoffman

Software Quality Management System **465**
Omar Alshathry

Supplemental Paper

Improvement Processes that Create High Quality Embedded Software **483**
Jay Abraham, Marc Lalo and Scott Runstrom

BOARD MEMBERS, OFFICERS, and COMMITTEE CHAIRS

Bill Gilmore – Board Member & President

Robert Cohn – Board Member & Secretary
Wacom

Doug Reynolds – Board Member & Treasurer
Tektronix, Inc.

Chris Blain – Board Member & Audit

Marilyn Pelerine – Board Member & Community Outreach
Symetra Financial

Paul Dittman – Board Member & Program Chair

Doug Hoffman – Invited Speaker Chair
Software Quality Methods

Ellen Ghiselli – Volunteer Chair

Shauna Gonzales – Luncheon Program Chair
Nike, Inc.

Rhea Stadick – Operations & Infrastructure Chair
Intel Corporation

ADDITIONAL VOLUNTEERS

Rick Anderson

Jon Bach

Bill Baker

Darrell Bonner

John Burley

David Butt

Moss Drake

Joshua Eisenberg

Brian Gaudreau

Cynthia Gens

Frank Goovaerts

Les Grove

Jacob Hills

Michael Larsen

Dave Liesse

Launi Mead

Bill Opsal

Dave Patterson

Ganesh Prabhala

Ian Savage

Jeanette Schadler

Wolfgang Strigel

Patt Thomasson

Mano Vela

Richard Vireday

PNSQC Online Community

PNSQC is now offering a new service – an online community! Our community **forum** is now 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 last year's conference survey 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. Recruits and works with incoming volunteers to place them in committees.

Project Patterns: From Adrenaline Junkies to Template Zombies

Tim Lister, Atlantic Systems Guild

Tim Lister and five of his partners at the Atlantic Systems Guild have compiled project patterns they've observed in their combined 150 years of project consulting and summarized them in a new book, Adrenaline Junkies and Template Zombies: Understanding Patterns of Project Behavior.

Tim will begin his keynote presentation with examples from his new book by describing common patterns he's observed at individual, project, and organizational levels. Tim has come to believe that project patterns are far stronger and more important than "best practices" will ever be. What are project patterns? They are the habits, decision-making practices, and unstated rules of corporate culture that dominate business life. The key to using project patterns is to identify your organization's current patterns. If they are positive patterns, how can you replicate them across all projects? If they are negative, how can you break the habits?

Tim's keynote will include audience participation! Tim plans to ask the audience to share some of their own project patterns. The goal is to go back armed with realistic goals and objectives for improvement in your organization.

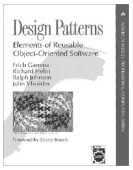
A principal of the Atlantic Systems Guild, Inc. based in New York City, ***Tim Lister*** *divides his time between consulting, teaching, and writing. Tim is a co-author with his Guild partners of Adrenaline Junkies and Template Zombies: Understanding Patterns of Project Behavior. He is co-author with Tom DeMarco of Waltzing With Bears: Managing Risk on Software Projects, the Jolt Award winner as General Computing Book of the Year for 2003-2004, and Peopleware: Productive Projects and Teams, now available in fourteen languages.*

Tim has over 35 years of professional software development experience. He is currently a member of the Cutter IT Trends Council, the IEEE, and the ACM. He is in his twenty-third year as a panelist for the American Arbitration Association, arbitrating disputes involving software and software services.

Patterns of Projects: From Adrenalin Junkies to Template Zombies

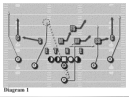
Tim Lister
The Atlantic Systems Guild Inc.

Patterns



Let's look at some Project Patterns…

More Patterns



Let's look at some Project Patterns…

Safety Valve

To counter the intensity of their work, the team devises a pressure release activity that becomes a regular part of team life.

Mañana

The loss of a "natural" sense of urgency.

The Dead Fish of Failure

It sits on the table of far too many projects.

How can we accept projects formulated to fail?

Everybody smells it right away.
Everybody hunkers down.

Who's to blame for this loathsome situation?

Me, and my generation.

The joy of success. It lets you:

- **try hard**
- **learn**
- **experiment**
- **have pride**

Lessons Unlearned

Retrospectives rarely trigger change. (Sorry.)

What Smell?

[redacted]

SMELLS…

Nursing Home
Monkey Cage
Teenager's Bedroom
Make-up Room Backstage
Sea Breeze
Mildew
Electrical Fire
Cigar Bar

Marilyn Munster…

The esteem often given technical workers versus managerial staff varies. In some organizations developers are kings; in others they are pawns.

Surprise!

Can you find the Big Boss in this picture?

The manager offering rewards and incentives gets responses in addition to those he planned.

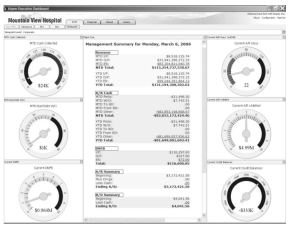
Project Sluts

Managers who just can't say, "No."

Rattle Yer Dags!

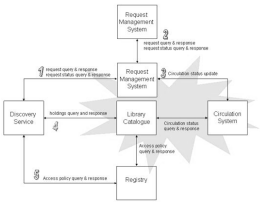
urgency + confidence + iteration = bent for action

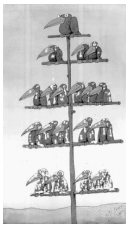
Dashboards



Dashboards are used by strong teams and weak teams, but typically not by average teams.

It's Always the God-Damned Interfaces





Project team members focus relentlessly on interfaces both system and human.

Cool Hand Luke

"What we have is a failure to communicate."

A legitimate conflict is interpreted as a "failure to communicate."

Straw-man

Team members feel comfortable offering a straw-man solution in order to elicit early feedback and insight.

Cider House Rules

Members of the project team ignore or work around rules made by people who are unconnected to the project's work.

White Line

The chalk line on a tennis court clearly defines the extent of the playing surface. The project needs a white line to achieve a non-arguable delineation of scope.

Endless Huddle

The right of infinite appeal ensures that no decision is ever final.

Music

People with real musical skills are disproportionally represented, sometimes extremely so, in technology organizations.

Look for Patterns…

Name them…

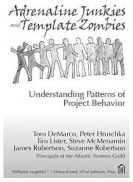
Propagate them, or

Defeat them.

You're good at this; you can do this together.

Send me your patterns!

Tim Lister
The Atlantic Systems Guild, Inc.
3143 Broadway, 2B
New York NY 10027 USA
212 620-4282
lister@acm.org
systemsguild.com



1-800 DH BOOKS: AJSPECIAL
www.dorsethouse.com/ajspecial/

Using Simple Automation to Test Complex Software

Harry Robinson, Microsoft

Software continues to grow more complex. Users want powerful features, they want those features to work smoothly, and they want those features delivered *yesterday*. Such demands make our industry a fascinating place to work, but they can make a test team's job a difficult challenge.

It might be natural to think that elaborate test systems will solve your problems; however those tools and infrastructures impose their own costs, and can distract you from your real mission of delivering great software.

Over the past year, Bing test teams have been experimenting with a *simpler* approach. Turning away from monolithic test infrastructures, we are finding that lightweight automation and heuristic oracles keep our tests flexible and productive while extending the reach of our exploratory testers.

Harry's keynote presentation will take the audience through the Bing team's journey to simplify and improve their testing — by offering the lessons learned, strange encounters during the process, and the encouraging results observed.

Harry Robinson *is a Principal Software Design Engineer in Test (SDET) for Microsoft's Bing team. Harry has over twenty years of software development and testing experience at AT&T Bell Labs, Hewlett-Packard, Microsoft, and Google, as well as time spent in the startup trenches.*

While at Bell Labs, Harry created a model-based testing system that won the AT&T Award for Outstanding Achievement in the Area of Quality. At Microsoft, he pioneered the model-based test generation technology, which won the Microsoft Best Practice Award. Harry holds two patents for software test automation methods, maintains the site www.model-based-testing.org, and speaks and writes frequently on software testing and automation issues.

using simple automation to test complex software

Harry Robinson
PNSQC 2010

Definitions

sim·ple
- able to be done or understood quickly
- lacking decoration or embellishment

com·plex
- difficult to analyze, understand, or solve
- made up of many interrelated parts

Encarta® World English Dictionary

Software: Complexity and Quality

"Software entities are more complex for their size than perhaps any other human construct …"

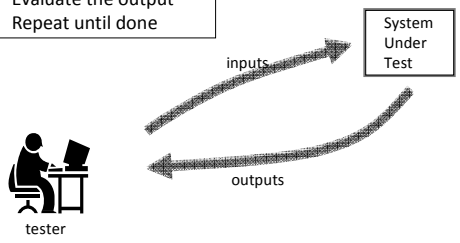
F. P. Brooks, **No Silver Bullet**

"Software complexity … grows to the limits of our ability to manage that complexity."

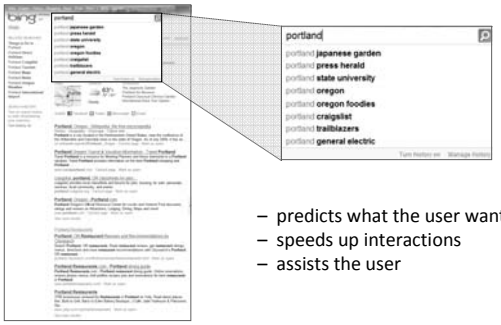
Boris Beizer, **Software Testing Techniques**

The Essence of Testing

1. Provide an input
2. Evaluate the output
3. Repeat until done



Welcome to AutoSuggest



- predicts what the user wants
- speeds up interactions
- assists the user

The Essence of Testing

1. Provide an input
2. Evaluate the output
3. Repeat until done

Input sources:

- static { d, de, def, stp, st p, portla, …}
- random {qxr, mxyzptlk, lfjgo, sj3g, …}
- exhaustive {aaaa, aaab, aaac, aaad, …}
- patterns {<popular queries>, <DSATs>, …}

The Essence of Testing

1. Provide an input
2. Evaluate the output
3. Repeat until done

Oracle sources:

- golden results
- eyeballs
- crashes
- heuristics

The Essence of Testing

1. Provide an input
2. Evaluate the output
3. Repeat until done

```
while (it's worth continuing)
{
    provide an input;
    evaluate the output;
    log the results;
}
```

Modes of Software Testing (1)

Manual Exploratory Testing

Pro
—Finds bugs
—Flexible

Con
—Labor-intense
—Slow
—Limited range
—Biased by intuition

Modes of Software Testing (2)

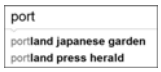
Static Test Automation

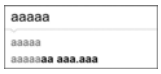
Pro
—Fast
—Repeatable

Con
—Rigid
—Fragile
—Limited input
—Finds few bugs
—Heavyweight
—Distracting

Good AutoSuggestions Are …

- Useful
  - Predictive
  - Fast
- Safe
  - No adult terms
  - No adult links
- Reasonable
  - No duplicates
  - No misspellings
  - No garbage text

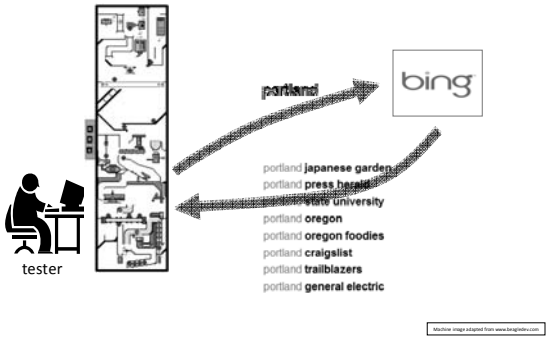




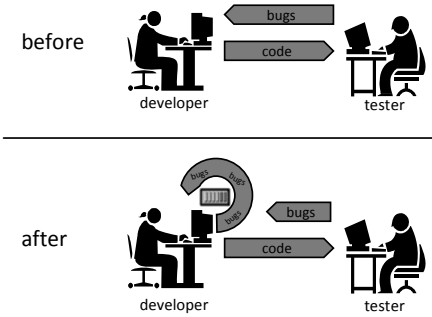
Functional Tests

#	Input	Expected output
1	d	dictionary, disney channel, delta airlines, dell, …
2	de	delta airlines, dell, detroit lions, derek jeter, …
3	def	definitions, defender, def jam, defraggler, …
4	stp	st pete times, stp, st paul pioneer press, …
5	st p	st pete times, st paul pioneer press, st patricks day, …
6	portla	portland japanese garden, portland press herald, …

Static Test Automation



Changing the Workflow



Heuristic Test Oracles

- Verify some individual results
- Check other results using
  - simple algorithms
  - consistency checks

Douglas Hoffman, **An Updated Taxonomy for Test Oracles**

Static oracle
sqrt(17) = 4.1231056

Heuristic oracle
sqrt(N) * sqrt(N) = N

Testing AutoSuggestion with Heuristics

Problem: How can we verify an enormous input space?

Input	Suggestion	Reason
def	def jam	prefix
nestb	bestbuy.com	spell correction
dond	nbc.com	?

Heuristic: Autosuggestions should be "explainable"

Results:

- Covered 22 million triggers in 7 days
- Detected 2000 bad suggestions

Emergent Test Opportunities

- Dead suggestions
- Suggestions with empty results
- Suggestions with adult results
- Misspellings
- Meaningless suggestions
- Powerful unit testing

New Risks

- Poor input generation:
  - insufficient coverage
- Poor oracling:
  - false positives
  - false negatives

Lessons Learned

- Simplicity can produce a virtuous cycle
- Oracles can be non-binary
- Humans and computers can coexist
- What seems simple may not be
- Simple != Easy
- Testing == Engineering

Acknowledgments

I am happy to acknowledge the work of Todd Paul, Reena Agarwal, Edward Unpingco, DJ Ramakrishnappa, Dragos Boia, Asraful Islam, Cyril Bouanna, Jing Tan, Andrey Yegorov, Nick Karnik and many other Bing team members in supporting, prototyping, experimenting with and developing the methods described in this presentation.

Contextually-driven System Architecture Reviews

F. Michael Dedolph

Abstract:
When the World Trade Center collapsed, the switching systems in the basement correctly diagnosed which lines were still working, and continued to connect calls using backup power for several days. One factor contributing to this remarkable product reliability was the AT&T / Bell Labs practice of early systems architecture reviews.

With concerns over systems and software reliability increasing every time you read the paper, some kind of architecture review is a necessity for any organization that wants to minimize liability while producing innovative, high quality products and services.

This paper will:

- Provide a simple model for defining and categorizing systems architecture that is context driven and representationally independent.
- Describe how to conduct an architecture review, using methods based on Lucent/Bell Labs Systems Architecture Review Board (SARB) process.
- Summarize the direct and indirect benefits of SARB-type reviews.
- Discuss how the review method was incorporated into the company's culture.

SARB-style reviews provide an alternative approach to the SEI's Architecture Tradeoff Analysis Method (ATAM) method. Compared to ATAM, SARB-style architecture reviews can be easily and flexibly tailored based on the context. The context for the review is established by the problem statement. The flexibility of the method makes it suitable for many kinds of systems and problem domains.

Background information:
The SARB review process was developed over time with extensive consulting support from Jerry Weinberg. F. Michael Dedolph was a SARB review leader for 7 years at Lucent/ Bell Labs, conducting more than 60 reviews during that time frame. The paper describes the review method as practiced at the time. Since the method is always evolving, current practices may differ.

Biography:
F. Michael Dedolph is currently a technical lead for a software process improvement effort within CSC. His organization recently achieved a CMMI Level 3 rating, and is moving on to Level 4.

From 1997 to 2004, Michael was a systems architecture review leader at Bell Labs (Lucent); he also managed and taught Lucent's Systems Architecture Introduction class. In addition to leading architecture Review Teams, he facilitated numerous risk identification, problem solving, and project retrospective sessions.

Prior to 1997, Michael worked in the Risk and Process programs at the SEI. While at the SEI, he was the technical lead for the teams that developed the SCE and CBA-IPI appraisal methods, and was the team leader for several Risk Reviews.

He started his IT career by spending 10 years as an Air Force computer officer.

What is “Architecture”?

If you want at least three different opinions about what an architecture is, ask any two architects.

Before we can conduct an architecture review, we need to know what we are reviewing. Unfortunately, there is little consensus in the computer and systems engineering fields about what architecture means.

To test this, ask a group of engineers or managers what architecture is, and record the answers. You will probably find design and interfaces mentioned a lot, but beyond that, very little real agreement.

For this discussion, the definition of architecture is that a system architecture provides a **solution** to a **problem** for a **Client**.

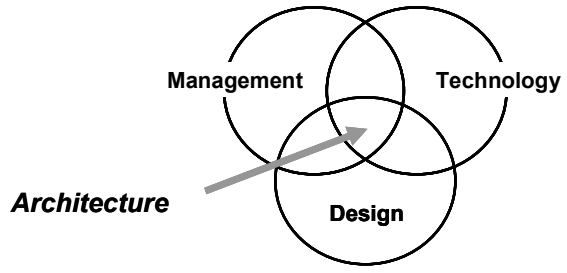
After construction, the architecture becomes the tangible solution you can see, along with a framework of supporting structures and system interfaces which may not be immediately visible. The constraints are part of the environment the system interacts with, and typically include operational performance and cost constraints.

Architecture has been characterized as “design with constraints”. Alternatively, you could say architecture *includes* a design that works *within* the constraints.

Before and during construction, the architecture is a *conceptual* or *potential* solution. During development, constraints also include cost and schedule.

Where can you find the system’s architecture?

Architecture exists at the intersection of management, technology, and design.



Management is concerned with cost, profitability, schedule, preserving legacy investments, customer willingness to pay, future liability . . .

Technology encompasses methods (process), materials, tools, approaches. Technology is constantly changing and evolving.

Design incorporates different views or models of the system for different users, includes interfaces, and, at the lowest level, must be “buildable”. Design concepts are often transferable across many domains and technologies. Because both architecture and design use multiple views, design is often mistaken for the architecture. For purposes of a review, different design views will be used to illuminate different aspects of the architecture, but design alone does not encompass the entire architecture.

Any given architecture is **not** the only solution, but some solutions are “better” than others.

How do we know how "good" an architecture is?

Taking a cue from the earlier question, ask the same group of engineers and managers what a "good" architecture is. This will likely produce even more varied answers, many of which boil down to "I'll know it when I see it". Other answers may focus on a particular aspect of the system, such as performance, reliability, or cost.

The key question for architecture reviews is how you can tell if an architecture is "good enough". To decide if a solution (or proposed solution) is good enough, first someone has to define what the problem is that the architecture is supposed to solve.

Problem Statements

Fundamental to this type of review is the concept that a system architecture provides a **solution** to a **problem** for a **Client**.

The problem statement establishes the context for the review by encapsulating the essential domain information, management concerns, technological challenges, and design constraints from the Client's point of view. The review method will examine the proposed or actual solution. However, to start the process, the Review Team needs a succinct problem statement that encompasses the problem.

A Framework for Architecture Problem Statements

The System Architecture Review Board (SARB) review method uses a specific framework for problem statements to address these aspects of the problem space:

- **Function** – what the system does, how "well" (fast, reliably, securely) it does it.
- **Form** – what it "looks like"; including the major environmental interfaces. Can be physical or logical form; for SW, includes things like protocols, languages, network environment, user interfaces, etc.
- **Economy** – cost and resource aspects, including development, maintenance, materials, system retirement costs, competitive pricing pressures, etc.
- **Time** – relationship to past, present, and future versions of the system, time to market, schedule.
- **Operational** – this is an extension to the original model. Operational aspects include things that pertain to development constraints and the operating environment, rather than the system itself.

The *function*, *form*, *economy*, and *time* (FFET) aspects have been widely used in civil architecture. Other frameworks exist[(1)]; this one was borrowed from Pena[(2)]. The framework was extended to include the *operational* aspects, because these aspects can affect the architecture and its supporting systems.

The different aspects of the problem space are overlapping and interrelated. Many aspects of the system will affect more than one area, and there will always be architectural tradeoffs between the different aspects. One goal of the review process is to make these tradeoffs visible and explicit.

Measurable attributes are desirable, but this may not be possible or practical for early reviews. It is more important is that the criteria can be judged. For example, if system performance will be measured as throughput under peak load, it will not be possible to measure the it until the system is complete. Early on, a performance model will have to suffice.

A quick example of FFET/O from classical civil architecture is the famous Taj Majal. A problem statement for the Taj Majal might have read something like this:

- The **function** of Taj Mahal is to provide a tomb for the Emperor and a monument to the Emperor's departed wife. It must be so beautiful that people will travel to it from around the world to honor her

memory, and know how much he cared for her. As a tomb for a Muslim Emperor and Empress, it must include a mosque on the grounds, and should provide a space for people to stay when they came to visit and marvel.
- The building **form** and materials will need to maintain symmetry, deal with the unprecedented weight of the structure, manage the shearing force of the nearby river on the foundations, avoid discoloration of the stone and fading of paint, and maintain visual perspective. A verse from the Koran will frame the main arch, and must appear to be written in the same size letters from the ground.
- **Economy** is unimportant. It will cost whatever it costs, and take as long as it takes. Up to 10,000 people can work on the Project, for as long as it takes.
- The structure is to last for all **time**. So the foundation must stand, and discoloration of the stone or fading of paint seen in previous palaces is unacceptable. Construction will take as long as it takes (work started around 1632 and was completed around 1653.)
- **Operationally**, there is insufficient expertise in the kingdom to build this. So the best Hindu and Persian architects will work together on the Project, and will hire anyone they need. Since some of the materials needed to meet the form constraints are heavy, a road will need to be built to transport them. Some valuable materials are needed that are only available in China, so reliable ordering and transportation must be arranged.

Against these criteria, the Taj Majal did pretty well. Construction was completed around 1653, and people still come to visit today, although they no longer stay on the premises. We might question the economic aspects of the problem statement—the eventual cost of the structure bankrupted the Kingdom and led to the overthrow of the Emperor.

Problem Statements for Reviews

The problem statement is the basis for the review—it provides a succinct summary of the critical success criteria. The problem statement includes major constraints, and covers the FFET aspects. Operational aspects may be included if needed.

A problem statement describing the necessary *function*, *form*, *economy*, *time* and *operational* (FFET/O) aspects of a system should:
- Be succinct.
- Provide the critical, discernible success criteria for the solution.
- Be expressed in sufficient detail to make judgments about the proposed solution, **but**,
- Avoid being unnecessarily proscriptive (“thou shalt not”) or prescriptive (“thou shalt”).
- Be client-centric.

Problem statements are **not** a requirements document, but may summarize the most critical high level requirements.

Problem statements also have unstated "always" criteria. These criteria depend on the Client and the type of system, and will include things like:
- It is possible to construct the system (operational).
- The Client can make money or will perceive value (economy).
- The solution won't result in harm (function, form).
- The solution is legal (function, form).

It is not necessary that the problem statement have separate sections for FFET/O, but it is often helpful. The important thing is that all of the aspects are covered in enough detail for the Review Team to use.

In addition to providing the criteria for the review, the problem statement establishes the specific expertise areas needed on the Review Team, and is used to establish the detailed review agenda.

The Architecture Review Problem

One of the things that makes the SARB-style review approach powerful is it's flexibility. To a large degree, the flexibility comes from the problem statement—if you can write a problem statement, you can review against it. To illustrate the flexibility of the problem statement concept, here is a problem statement for an Architecture Review method.

==========

Architecture Review Problem Statement:

The review method should increase the probability of success for the Project (function) by addressing FFET/O aspects of the system being reviewed (form). Findings generated by the review need to be unbiased, independent, and relevant for both management and technical staff (function). The findings must be complete enough to alert management to high-risk issues (function), but also framed in a way that promotes project level "buy-in" for resolving the issues (operational, functional, form.)

The method should consider design, technology, and management constraints (form, operational), be representationally independent (form), and flexible enough to work in multiple domains (form, function).

The reviews must be cost effective (economy), and whenever possible, they should leverage existing expertise within the company (economy).

The review method should work at any point in the lifecycle, but, in particular, support early reviews (time, form). 90% or more of the reviews will be conducted in 3 days or less by a small, in-house team of 3-10 people (time, economy).

==========

As we discuss the SARB method, you can review it against this problem statement to see how well the method meets the need. Before proceeding, though, you might ask "Is there anything missing in this problem statement?", and "Are there any aspects of the problem I don't understand"? (These questions would typically be addressed in the pre-review meeting early in the process).

Fundamental Architecture Review Method

In a nutshell, the review method is:

- Define the problem the Client wants solved,
- Compare it to the architecture (proposed solution),
- Identify the gaps (or risks).

After the review, let the Project resolve the gaps.

The first activity is done by the Project and Client, possibly with consulting help from the Review Team. The next two activities are done by the Review Team. The project owns resolution of the findings.

Architecture Review Questions

Here are some key questions that need to be answered as the review unfolds.

Questions to answer **before** the review by the Client, project, and Review Team:

- Who decides what the critical success criteria are, and who sets priorities? (Know the Client)

- What problem are we trying to solve? Do both the Project and the Review Team understand the problem?
- Are key stakeholder interests represented?

Questions for the Review Team to answer **during** a review:
- How good is the proposed solution?
- What are the issues/ risks/ gaps in the solution?

Questions for the Project to answer **after** a review:
- Which things will we address?
- How will we address them?

How to Conduct an Architecture Review

Before listing the steps in the steps in the SARB review process, we'll briefly cover the roles in the process and the ground rules for a review

Review Roles

For the Review Team, essential roles in the process are the Review Leader, Angel, and Review Team Member. All reviewers are independent of the Project's reporting chain. At Lucent, the review **Angel** was a SARB Board member.

The **Angel** represents management interests for the company. In this role, the Angel co-facilitates the review activities, helps with recruiting team members, ensures follow up actions are appropriate, reviews the reports, and facilitates transfer of lessons learned to other projects. The **Review Leader** coordinates the review logistics, consults with the Project on the problem statement, recruits Review Team members based on the problem statement, facilitates the review activities, writes or edits the final reports, and coordinates follow up activities. Review Team **Members** bring the specific expertises to the table that are needed to cover all aspects of the problem statement.

For the Project, the **Client** is the person paying for the Project and sponsoring the review. Ultimately, the Client is responsible for the success of the product and project. In this capacity, the Client makes final decisions about what the critical success criteria for the product and project are, and decides how critical tradeoffs will be made if all the criteria can't be met. Typically the Client is at the VP level or higher. Using the Client as the decision maker is an implementation of the corporate "Golden Rule" – the person with the gold makes the rules.

The person designated as the **Architect** assists in preparing the problem statement, and is responsible for arranging the technical presentations used during the review. They typically present the top-level architectural view of the system. The architect usually has responsibility for following up on technical review findings. Architects may be part-time staff on small projects, or a part of an architecture group on a large project. For architecture groups, the lead Architect will coordinate review activities.

Project Management works with the Client and Architect to prepare the problem statement, and presents the problem statement and project overview to the team. They also assign some one to work the review meeting logistics, and are responsible for managing follow up actions after the review.

Other **project participants** include the presenters who have expertise in particular areas. Projects may choose to keep review involvement limited to the manager, architect, and presenters, or may use the review as an educational opportunity for staff members to get up to speed.

Review Ground Rules

Ground rules are covered in the pre-review, and again at the start of the review meeting. These ground rules are based on time-honored review principles established by Weinberg[3] and others.

Reviews should be done without attribution—review products, processes, and ideas, not people. Frame issues and observations in terms of the product architecture and the problem it is intended to solve, not in terms of the presenter or architect

The Review Team is there to identify, not solve problems. (Asking engineers to follow this ground rule is a bit like asking cats not to shed on the couch. However, you can train engineers. Ask them to write their on-the-spot ideas as observations or suggestions, and record them on an observation/suggestion card.)

Review and project team members can ask questions at any time. The speaker may defer the answer, and the Review Leader may ask that the question be held until later. But, the basic rule is, ask it.

Review Team members should write any notes, questions or thoughts they have on cards. At the end of each presentation, Team members review their cards to make sure their questions got answered. If not, they should tag the card as an issue, or keep it as a question to follow up on later.

The Client requests and pays for the review (in the sense that the Project team is spending time on the review), but the Project owns the findings and responsibility for correcting them. There is one exception to this rule, the “management alert” (covered later).

In summary, as reviewers, the goal is to do no harm—to paraphrase the House of Blues slogan “help ever, hurt never”.

Review Phases and Steps

The review is broken up into three chronological phases: **Preparation**, **Review Meeting**, and **Follow-up**.

Preparation steps

1. Respond to initial contact/request.
2. Develop the problem statement.
3. Select team and develop agenda based on the problem statement.
4. Arrange logistics – travel, space, phones, connectivity, etc.
5. Hold pre-review call.

Review Meeting steps

1. Review the Project’s architecture presentations, with Q&A .
2. Hold Review Team Caucus.
3. Conduct Readout.
4. Hold Sponsor/Client meeting (optional).

Follow-up steps

1. Write and distribute the Review Report.
2. Present findings to lessons learned/review process governing groups (Board Report).
3. Review the Project’s action plans.
4. Close the review by meeting with the Project team and/or Client.

The discussion that follows covers the review activities, by phase. Not every step is discussed, but some steps or activities are called out in detail because they can be problematic, or because they represent something relatively unique to the method.

Preparation Considerations

Step 1 – Responding to the initial contact/request is essential for setting the right expectations for the review, especially for teams or organizations that have never been part of a SARB-style review.

Step 2 – Client and Project team develop a Problem Statement. The Problem Statement is the basis for the review. It seems like this would be easy, but developing it and getting agreement to it is often the hardest part.

In some cases, projects canceled a review because they didn't have the time to develop a problem statement. (In the author's experience, these projects were not often successful).

In many other cases, the initial review schedule was delayed while the Project and Client worked out the details of the problem statement.

One of the tough parts seems to be making it short. There seems to be great temptation to hand off a big honking requirements document in lieu of a succinct statement that reflects the truly critical success criteria. There is good reason for insisting on a short problem statement—the Project team members are likely to keep the short "vision" for the product in mind, but will only reference the requirements documentation if they need to work with the details.

Step 3 – Team selection is based on the problem statement. If the product has stringent performance, reliability, or security criteria, then you need that expertise on the team. Conversely, if the product doesn't have a critical data base component, don't bring in a data base expert.

To keep the team size small, senior people who can fill in multiple areas are desirable. Over time, the review leaders will become sufficiently rounded to cover (or be the backup) for most areas in the domain, and will recognize common pitfalls. In general, having more than one or two junior people on the team is detrimental.

Step 5 – The Pre-review call is a critical step to ensure that the Project and Review Team members are all in synch. The call serves as "just-in-time" training for new Review Team members, and for Project Team members that have not participated in a review.

During the call, the Project and Review Teams will go over the ground rules and review conduct.

Next, there will be a fairly detailed look at the problem statement. This is the Review Team's chance to ask questions and clarify the problem space, and to identify possible omissions.

- ***What do you mean by ...?***
- ***Did you consider … ?***
- ***Is X a factor for your system? Should it be in the problem statement?***

For example, in the Architecture Review problem statement, the phrase "using a small, in-house team" might lead to questions like: (1) How small is small?, and (2) Does in-house mean the team can include Project Team members? The answers to these questions could lead to a modification of the problem statement, or to a better understanding of the meaning.

After the review of the problem statement, the team and project will verify that the agenda covers the problem statement topics, and that sufficient time has been allocated to cover the topics.

The last step is to request pre-reading materials, and to arrange for distribution to the team members. During a 3-day review, the Review Team will not get to see all of the details of a reliability model, but the reliability expert on the team can look at the model in advance to frame questions for the review.

Review Meeting Considerations

For SARB reviews, the review meetings were conducted face to face. In rare cases, an experienced reviewer was allowed to participate virtually; this is not as effective.

Virtual reviews are not as effective. This was (and is) increasingly problematic because of travel costs and extensive use of distributed teams.

Project Team member presentations can be done remotely, but a few Project Team representatives should be face to face with the Review Team. Typically the senior Architect and either the PM or deputy would attend the meeting. However, projects should keep in mind that a face-to-face review can have other benefits such as team building and team training that go well beyond the findings.

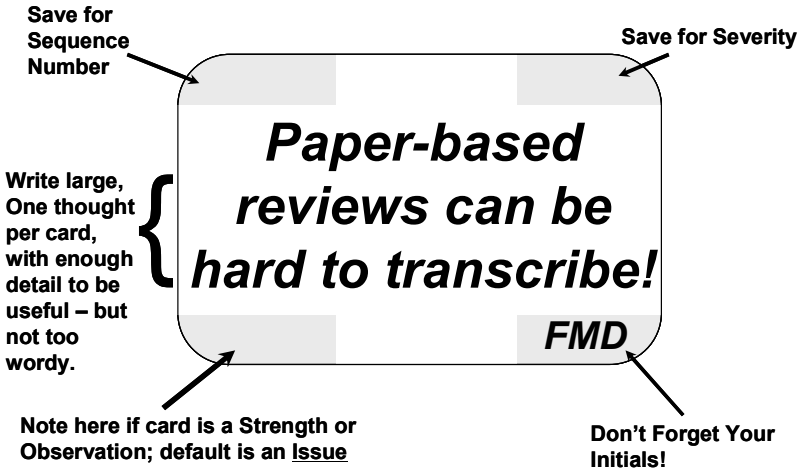
Step 1 – Review the Project's Architecture Presentations, with Q&A.

Presentations are made by the Project team members in any format. PowerPoint can actually be detrimental and whiteboards can be good. As a rule, project should **not** develop totally new materials for the review, unless there was a major gap in planning (oops, no security model yet, better get a draft ready for the review). Usually existing presentations can be adapted for the review.

Questions can be asked by the Review or Project Team members at any time. The discussion is moderated by the Review Leader or Angel if needed.

During the presentations, strengths, issues, and (optionally) observations or suggestions are recorded. For SARB reviews, specialized index cards called "Snow Cards" are used to keep notes. Both review and project team members can write cards. The cards document findings in real time; editing and analysis is done during the Review Team Caucus.

Snow Card Example



Snow cards were a tradition at AT&T and Lucent, because they are ideal for the purposes of a review. A large index card or Post-It would also work. For some reviews, the paper based snow cards were supplemented by virtual "cards" collected in a data base. New tablet based media would also work.

However, in order to rapidly group, categorize, and prioritize the findings from the data base, the virtual cards had to be printed.

Like any technology or design feature, snow cards represent a trade-off. The visible **benefits** are that paper-based snow cards provide a real-time, unobtrusive way to capture issues and ideas (keyboards can be a major distraction). The cards can be posted on a wall, providing instant visibility into the findings. The media also supports being able to quickly edit, group, and prioritize the issues. Other, less visible benefits of using snow cards (versus laptops) include keeping the team focused on the review presentations rather than a pop-up e-mail, and keeping the findings relatively short. The **drawbacks** to using cards are that they need to be transcribed for the report, they can be overly terse, and they can be hard to read.

Step 2 – Hold Review Team Caucus.

The Review Team Caucus is led by the Review Leader and Angel. Only Review Team members participate in the caucus. During the caucus, the Review Team categorizes, summarizes, and prioritizes the findings. After analysis, the findings are then used to prepare the readout for the Project Team, and from there flow directly into the Review Report.

A “typical” review produces 80–350 observations, arranged into 10-25 topics. **Categorization** is done by the reviewers as a quick group exercise, using affinity grouping. Each affinity group is given a topic heading. The topics are created to provide a consistent “story” for the read-out. Most of the topics will map to the review agenda or to problem statement areas.

Strengths and issues may be categorized separately, or included in each topic area with the issues.

During readout prep, a **summary** of each topic area is written by the Review Team subject matter experts, working alone or in small groups. This can be done by hand on the back of a snow card, transcribed into PowerPoint for the readout, and then re-used for the review report.

Prioritization can be at the level of a topic area, a subset of snow cards within a topic area, or an individual card. Strengths and Observations/Suggestions are not prioritized.

Since reviews point out issues that often lead to architecture changes, it is important to point out the things that are going well. **Strengths** are used to document architectural or project features that should be retained going forward.

The review process de-emphasizes making suggestions, because it gets the Project team into the mode of problem solving. However, some suggestions are inevitable, and they can be useful. For example, a suggestion to “Contact Susan in the XYZ project about their approach to optimizing data base performance for open source data base systems” could be very handy when the Project addresses related issues.

The SARB review process uses these priority categories for issues: **management alert**, **critical**, **major**, and **minor**.

A **management alert** is sent to the Client and/or the Project’s upper management, and represents the one exception to the rule that the Project owns the findings. All other findings go to the Project—critical, major, and minor issues, along with strengths and observations.

Determining that an issue is a management alert requires team consensus. If the Review Team does not reach consensus, the issue is categorized as a critical issue. The criteria for deciding an issue is a management alert is found in the header used to present the issue: **“In the unanimous opinion of the Review Team, the Project will <u>fail</u> unless these issues are immediately addressed:”**

Unlike a management alert, the critical, major, and minor priorities are set by the team member who is the subject matter expert in the area, or by the Team Leader.

The criteria for deciding that an issue is critical are found in the critical issue header: “Unless these issues are addressed, the architecture will not meet all of its success criteria, resulting in significant rework and/or customer dissatisfaction. These issues are seen as complex, and/or will require significant effort to resolve.” Major and minor issues represent a partial failure to meet the Projects success criteria, but are typically seen as easier to fix.

Step 3 – Conduct Readout

The readout shows the Project what the most important issues are, so they can get started, and so there are no surprises in the Review Report. Upper management (e.g., the Client) is invited by the project, only if the Project wants to include them in the readout. Readouts use a PowerPoint presentation to summarize the snow cards, supplemented by the snow cards themselves.

The Review Team will **<u>not</u>** withdraw a finding or lower the priority. Since the Project owns the findings and responsibility for resolving them, if the Review Team over-reacted, the Project can adjust when they do action planning.

However, an issue's priority may be increased at project request. In a few cases, project teams explicitly asked that a critical issues be turned into management alert issues, because they had been trying to resolve the issue and needed upper management support.

In some cases, the Client/Sponsor will request a supplementary meeting to provide a private forum for discussion.

Follow Up Considerations

The readout and Client meeting are the last steps in the review meeting, but can also be thought of as the first steps in follow up. A copy of the readout is provided to the Project so they can immediately begin action planning to resolve the issues.

Management Alert and Critical Letters are sent within a week. Management alerts go to the Project's upper management and/or the Client (usually the Client is part of the Project's upper management chain). The Project's upper management is responsible for the action plan. Most management alerts are related to cost, schedule, and resource issues. Only rarely will a management alert be purely technical in nature.

The Critical Issue letter goes to the Project, and Project management is responsible for the action plan

The review report includes a transcription of all snow cards, along with summaries for all the issues. It provides the Project with a level of detail to support action planning, including the suggestions form the team. Reports should be completed and sent within 2 weeks.

If there is a neutral oversight organization, a separate Board Report will be generated to provide a corporate-level summary and to capture lessons learned.

For management alerts and critical issues, there is a specified time window for the responsible manager to prepare an action plan. Action plans are **not** required for major and minor issues. Although the Review Team has no ability to enforce the action plans, the plans are reviewed by the Review Team, and feedback is provided to the Project. **Closure meetings** with project and/or sponsor are used to provide feedback to the Project after the action plans are reviewed.

Project Ownership of the Findings

Project ownership of the findings is a key feature of this type of review. Project responses to a review range from putting the findings in a drawer and refusing to participate in action planning or any closure activities, to posting all of the findings on the Project's web page along with the action plan. (Guess which was the more successful approach).

Most projects do **not** address all of the findings – there simply isn't time, and the snapshot taken by the Review Team gives way to new challenges as the development activities unfold.

Given that addressing findings is at the discretion of the Project, what happens:

- When a project ignores a Management Alert or Critical issue?
- When a project fixes everything?
- When the Review Team identifies an issue that turns out to **not** be a problem ("false positives")?

Ignoring Issues

Experience shows that ignoring a management alert or critical issue is not a good path for the Project to follow. Without naming the projects, here are two examples.

One project tolerated an architecture review they didn't want, then used their prerogative to ignore several management alert and critical issues. The project later became a textbook example used in internal classes about what not to do, while losing the company a **lot** of money (not millions, think big).

A more typical and prosaic example saw a project neglecting a critical performance issue because it was hard to resolve, and the schedule was tight. The issue identified a mismatch in interface as a potential performance bottleneck. Ultimately, the system hardware met the specifications, but the software protocol mismatch caused the equipment to handle only 50% of what was required. This meant the customer had to buy twice as much equipment to achieve the performance they wanted, and/or the product price had to be discounted sharply. The result—major dissatisfaction for everyone.

Fixing Everything

As to fixing everything, the author never witnessed it. At least one proactive project had annual reviews for each major release, posted the findings on the Project web site along side the action plan, and worked off all of the Management Alert and Critical issues. The product continued to improve over time.

The stellar reliability and performance of the 5ESS switch was mentioned in the abstract. These results were due in large part to a culture of reviewing every major change, and resolving the issues that were identified. In addition to architecture reviews, extensive use of software peer reviews and several phases of testing contributed to the product's legendary reliability. As part of every architecture review, the problem statement addressed performance and reliability issues. By the way, this example shows that the utility of reviews is **not** limited to new products—in fact, it may be more difficult to act on review issues for new products may because of market time pressures.

Issues that Aren't Really Issues

Identifying issues that turn out **not** to be a problem is not really a disadvantage of the method, if you take a risk management perspective. A "false positive" forces the Project to do due diligence on the solution; the Project's action plan may simply state: "Here is the issue, we looked into it, and our original approach is the best available option at this time."

Major Benefits of this Type of Review

SARB review sponsors included people who were clients for reviews—project managers, directors and VPs that had come to see the value of the process.

A survey sent to the SARB review sponsors [(4), (5)] found that reviews had the following effects—projects avoided problems, identified issues, and as a result, saved money.

Savings from risk and issue avoidance are hard to quantify, but it was estimated that the savings averaged more than $1M per large project, especially if reviews are done early[(4)]. Similar estimates were derived from internal Monte Carlo simulations based on analyses of architecture review findings[(4)].

In some cases, there were direct results attributable to a specific review. One sponsor told his team to re-engineer the product after a review. The resulting architectural changes trimmed 9 months off the development schedule and saved more than $7 million.

Sponsors also noted less tangible benefits, beginning with the review preparation. Activities such as preparing the problem statement forced people to think through the problem, and communicate their thinking with other team members and management.

Sponsors also noted cross-pollination of techniques and practices across the larger organization. This was echoed by Review Team members, who enthusiastically cited how much they learned while participating.

Other benefits noted by some sponsors included synching up Project Team members with work others on the Project are doing—getting everyone "on the same page".

Another, longer range benefit was the ability to focus and refine a set of key success factors for a product family over time, e.g., reliability improvements in the 5ESS switch.

Potential Shortfalls of this Type of Review

SARB-style reviews are a potent tool for improving systems and software quality. However, like any other technology or method, there are trade-offs that can represent drawbacks.

Risks of the Client Centered Approach

There is a fundamental risk inherent in the client-centered approach. The advantage is that responsibility for decision making is clear, and the funding source is plugged into resolving the issues, which minimizes politics. However, for a variety of reasons the Client can grow out of touch with the market, customer needs, and end user issues. Making sure that alternative points of view are heard by the Client can be a problem. If these issues are apparent, the Review Team (particularly the Angel) can raise the issues when the problem statement is being clarified.

In government contracting, many of the success criteria are negotiated into the contract, and represent political compromises rather than a specific, knowledgeable decision maker's opinion. The government contract officer is the closet person to a "Client", but typically is a contracting specialist rather than a knowledgeable system user or engineer. The contract officer may not be willing or able to make decisions that contradict the "black and white" of the contract.

Incomplete Findings and Level of Detail

The review findings are inherently incomplete. Any finding identified early is a plus for the Project, but the issues found in a review will always be missing something—hopefully something with a lower impact on project success. This is true of any method, but provides a cautionary note for using SARB-style reviews as the only mechanism for identifying issues.

Although the problem statement can be focused on particular, very detailed aspects of the system, the review technique is best used when the goal is to make sure the whole architecture hangs together. But, a high level review is not suited for all problem spaces, and 2 to 3 days will **not** flush out all of the "devils in the details".

For example, a high level review may identify that the performance model for a subsystem is incomplete because it omits a major interface, but is unlikely to identify a specific problematic transaction for the system. The rework done by the Project to update the model at the detailed level is much more likely to catch the problem transaction, so the review can facilitate correcting the details, even if they are not caught in the review.

Resistance to Face to Face Meetings

There is growing resistance to corporate travel, so it is increasingly difficult to arrange the face to face environment where SARB-style reviews are most effective.

What's Different (or the Same) About this Kind of Review?

In common with other review methods such as the SEI's ATAM[(1)], SARB-style reviews bring impartial, outside eyes into the Project by using a team that is external to the development effort. Like other walkthroughs and inspections, products, processes, and ideas are reviewed, not people. Also, the review is **not** a problem solving session or fact-finding audit.

SARB-style reviews differ from other review methods in the following ways:

- **The method emphasizes problem and solution congruence**, rather than adherence to a particular notational standard or concept of what an architecture should look like. For example, a system may be described as a Client-Server or Service-Oriented Architecture, but the system as defined may not solve the real problem.
- The **Client has ownership** of the decision making process.
- The method is **context and domain independent**—the problem statement is used to adjust the review to the domain.
- The method is also **notationally and representationally independent**, not model or standard based, although these aspects can be included in the problem statement, if needed. In other words, you don't have to do an object oriented design if a state machine or transaction model is a better fit.
- Follow up includes a **clear statement of potential failure areas**, with a prioritized hierarchy of findings.
- **The Project sees and owns the findings**. The Project can throw them away, or post their action plan to their web site. There is balanced but extensive **project participation** during the review, and the Project Team is trusted to be responsible stewards of problem resolution efforts afterwards. All of this helps to build buy-in.
- The role of **Review Angel** is unique, and helps the Review Team focus on providing an unbiased look at the technical and management aspects of the architecture without trying to deal with any political fallout.

In summary, SARB-style reviews balance a defined review process with extreme flexibility.

SARB-style Reviews and ATAM

In the Author's opinion, SARB-style reviews are more flexible because of the use of the problem statement, and the reliance of ATAM on particular architectural representations. The free-wheeling nature of SARB is better suited for early concept exploration, and SARB-style reviews work for various kinds of systems and domains—hardware and networks, as well as software.

SARB reviews take less time, and hence they probably cost less.

ATAM reviews provide more structured outputs that can be revisited systematically over the life of the Project. In particular, the ATAM focus on identifying explicit risk trade-off points for the software architecture provides a way of evaluating the impact of changes over time without re-reviewing the entire system.

There is some convergence— both SARB-style reviews and ATAM can be used for risk identification and mitigation[(6)]. Also, recent changes/additions to the SEI's Architecture methods have focused on defining architectural attributes that are explicitly related to business goals[(7)], which is reminiscent of the concept of a problem statement. The SEI also recently extended the ATAM method to include systems as well as software[(8)].

In fact there is room for both—they could be used as complimentary techniques. For example, maximum benefits might ensue if an early SARB-style review at the systems level was followed by an ATAM review of the software after the high-level design was complete.

Incorporating reviews into the organization's culture

To close, here are some notes on how the process became part of the corporate culture. These notes reflect the author's experience at Lucent, and probably do not reflect the current environment.

Architecture reviews were embraced enthusiastically by some parts of the company, and rejected by others. The same pattern held with companies acquired by Lucent—some embraced the reviews, others fled from them.

Initially, a particular Vice President who was close to retirement adapted the SARB review methodology, and promoted it as his legacy to the company. He was a relentless champion of the method and paid to establish a core group of review leaders, and, with the advice of a consultant, worked to establish the SARB Board.

The SARB Board provides oversight of the method. Essentially the SARB Board is a large, distributed group of review sponsors. Board members have multiple roles—they serve as review Angels, act as Clients for their own projects, promote the method, share lessons learned, and provide team members and travel funds to support reviews on a quid-pro-quo basis.

The initial funding model for SARB was central funding by one VP, which evolved into corporate level central funding by the Chief Technical Officer. As times got tight in the Telecomm industry, the funding model changed to funding by Sponsors. Sponsors set aside a portion of their budgets to pay for a certain number of reviews in their organization each year; this money was used to retain a core team. Excess capacity in the core team was used to conduct a few other reviews for high visibility projects. Other reviews were funded on an as-requested basis by the requester. Typical cost of a requested review was in the $40-70 K range (cheap compared to the potential $1M savings).

Project ownership of findings coupled with senior management insight into critical problem areas has proven to be a good basis for project's wanting to participate, providing a "bottoms up" push for reviews in some groups.

As a strategy for other organizations, the following steps may serve as a starting point:
1. Find a high-level champion.
2. Obtain central funding for an extended period of time – say 5 years.
3. Train a core team of leaders and angels to do reviews.
4. Charter a board; recruit advocates.
5. Perform the reviews; capture data and lessons learned to document the method's value.
6. Move to a sponsor-based funding model after the initial time period, when the method has proven itself to the Project teams and sponsors.
7. Retain the focus on client-centered problem statements, project ownership of findings, limited but pertinent management alerts, and non-attribution.

References:

1. CMU/SEI-2000-TR-004, "ATAM: Method for Architecture Evaluation";
Kazman, Klein, Clements, 2000

2. Problem Seeking: An Architectural Programming Primer, 4th Ed;
Pena and Parshall, 2001, ISBN 0-913962-87-2

3. Handbook of Walkthroughs, Inspections, and Technical Reviews;
Freedman and Weinberg, 1990, ISBN 0-932633-19-6

4. IEEE Software, March/April 2005, "Architecture Reviews: Practice and Experience";
Maranzano, Rozsypal, Zimmerman, Warnken, Wirth, and Weiss

5. STQE Jul/Aug 2002 (Vol. 4, Issue 4) Feature: Measurement & Analysis
"A Blueprint for Success: Implementing an architectural review system";
Daniel Starr, Gus Zimmerman

6. CMU/SEI-2006-TR-012, "Risk Themes Discovered Through Architecture Evaluations"; Bass, Nord, Wood, Zubrow; 2006

7. CMU/SEI-2010-TN-018, "Relating Business Goals to Architecturally Significant Requirements for Software Systems"; Clements, Bass; 2010

8. CMU/SEI Webinar, "SoS Architecture Evaluation and Quality Attribute Specification (Webinar)";
Gagliardi; 2010

Effective Testing Techniques for Untold Stories in Story-Driven Development

Erbil Yilmaz, Gokhan Ozer
erbily@microsoft.com, gokhano@microsoft.com

Abstract

Feature crews (teams) take a few stories and implement them as part of story-driven development in each sprint. The goal, at the end of each sprint, is to demonstrate and complete the customer experience along the stories designed. This is a great software development methodology proving the software being built can deliver the customer experience.

However, story-driven development brings unique challenges for software testing. As stories are implemented, relying on each other, a complex software system emerges capable of doing more than just the stories told. With the addition of each story, testing surface and capabilities (and defects) of the software grow exponentially.

This paper describes various techniques developed and used over several releases through experiments in testing within agile development by the Visualization & Modeling Tools team for Visual Studio Ultimate. It also presents a case study to illustrate and emphasize key points, ranging from using architecture diagrams as part of test planning to particular test coverage along with test spectrum range. The guidance presented here focuses around a set of best practices that teams can adopt to accomplish optimal test coverage over the test spectrum.

Biographies

Erbil Yilmaz has over 7 years of software testing experience with Microsoft as the owner of security and performance exit criteria for the Visualization & Modeling Tools team for Visual Studio Ultimate. He is part of the Test Architecture Group and drives integration testing with agile development, effective test planning, and performance testing strategies. He holds an M.S. in Computer Science specializing in Computer Security from Florida State University. He also has B.S. degrees in Computer Engineering and Mathematics from Bogazici University.

Gokhan Ozer has also been working on Visualization & Modeling Tools team for Visual Studio Ultimate for 2.5 years, as the owner of stress/memory exit criteria. He holds an M.S. in Computer Science from University of Houston and B.S Computer Engineering from Istanbul Technical University.

1. Introduction

Agile methodology focuses on producing working software that can be demonstrated (close to shipping) to customers at the end of each sprint. The software product is in increments of enabled stories or features over multiple sprints; inherently all software development activities required to ship the current product are completed within the sprint boundaries.

However, this brings several unique challenges to software testing:

(1) Usually, different parts of the software are developed by different sub-teams, making it very difficult to manage and understand a complete set of capabilities of the software. As a result, at the end of each sprint, the software product consists of:
   a. Features that are developed for the told stories
   b. Features/defects that emerge from untold stories, which result from the interactions of the implemented stories. Sometimes these features are as designed, but often hide significant integration bugs.

(2) Focus on testing the story within the iteration accumulates test debt for integration points that is hard to estimate and requires long stabilization/verification cycles after all stories are implemented (code-complete).

(3) As iterations progress, more test defects will be discovered around the integration points (untold stories) rather than coming from existing test coverage. This causes questions around the existing test coverage and test automation, if it exists.

These challenges are not solved yet, partially because software testing has not matured enough within agile development methodologies. This paper introduces a set of techniques and best practices that were developed by the Visualization & Modeling Tools test team over the last two releases to alleviate these challenges.

2. Testing for Stories in Agile Methodology

In Agile methodology, stories define what the customer will be able to do after the implementation of the feature. Naturally, test planning and testing for a given story focuses on variations and contexts around the story, enumerating different use cases within the scope of the story in test plans.

There is also significant focus on unit testing and acceptance testing as part of iterations. While unit tests ensure that implemented methods achieve what they are designed for, acceptance tests target the end-user experience at the system level. In both cases, there is a test design tendency towards isolating the software implementing the story from its dependencies or interactions with other features with various techniques such as mocking interactions.

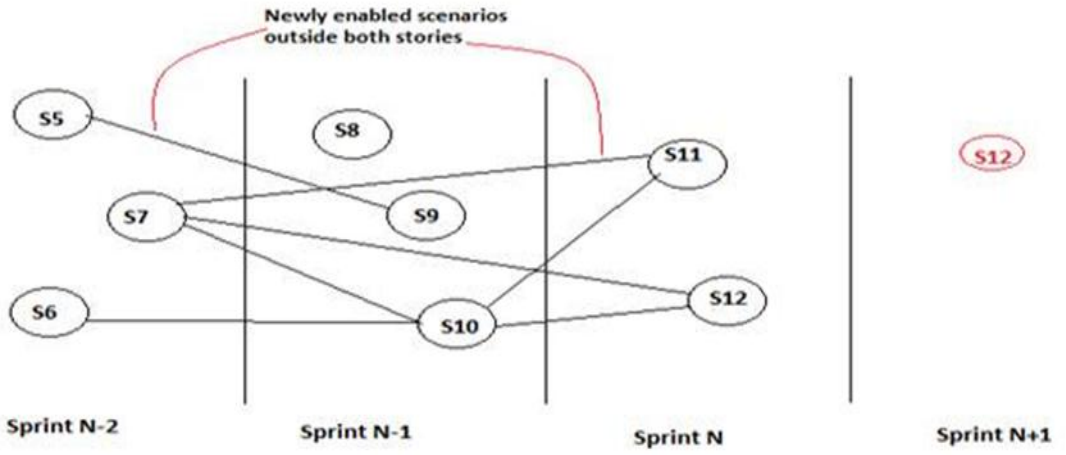
As a result, for each sprint, stories become high level encapsulations of software increments fulfilling stories (features), and test cases mostly verifying stories in isolation.

3. Testing Challenges arising from Story-Driven Development with Agile Methodology

Story-driven development over iterations brings several challenges to testing. As iterations progress and new stories are completed, new test surface emerges from interactions of stories that inherently are not well covered with testing focused mostly on stories within iterations.

The following figure explains how interactions between stories create a complex map of stories over iterations. Each link between stories represents a newly enabled scenario that is not accounted for in either story. This section will describe a series of testing challenges with references to this figure. As a result, for each sprint, stories become high level encapsulations of software increments that fulfill stories (features), and test cases that mostly verify stories in isolation.

For example, in the following diagram, S11 is an extended story on S10, but, through implementation, it also enabled new scenarios interacting with S7 that were not covered with S7 or current S11 test planning.



3.1 Complex Story Map over Iterations: Untold Stories

Discovering a complete map of interactions between stories is a very hard task, which is proven by the significant number of defects that are found after iterations and that are not covered by designed test cases. However, carefully examining these interactions to uncover defects could be overwhelming. In some cases, interaction between stories might be easy to discover if the current story is a natural extension of previous story in consecutive sprints. However, if the stories are not in consecutive sprints, or if multiple stories are involved in interactions such as through a cross-cutting feature, it is extremely hard to plan tests around these interactions. This results in incomplete test plans with potential high risk defects.

3.2 Integration of Features Developed by Different Teams

If there are multiple teams involved in developing a set of stories that contribute to a feature, the chances of discovering all interactions decrease significantly, and assumptions and code interactions are not surfaced.

3.3 Accumulated Test Debt

Since most of the interactions among features are not covered within sprints delivering stories, teams usually accumulate significant test debt over the release cycle. As untold stories are discovered, new test work items are added to the product backlog. Even though defining newly found test work helps to ship the product with fewer integration defects, it doesn't solve all the potential problems. Additionally, the product owner is likely to continue to prioritize new features over older test needs (or debt).

Depending on the time of discovery in the product cycle, teams react to these test holes in two ways. If the integration test hole is uncovered during coding sprints, then agile teams usually expand test plans to cover these areas and implement regression test cases for all significant bugs. This is accounted for as newly found work and treated like the rest of the backlog items. Even though this does not disturb the rhythm of sprints, it still extends the product ship cycle and questions the readiness of the implemented feature. In some cases, it requires architectural changes with effects on the rest of the features or cross-cutting features.

In general, fixing defects that are discovered later in the product cycle costs more. If these inherit integration defects are discovered late in the product cycle, it becomes even harder to react. In some cases, untold stories are discovered after the release by customers causing expensive patches or service packs and more importantly the loss of reputation and lost future sales.

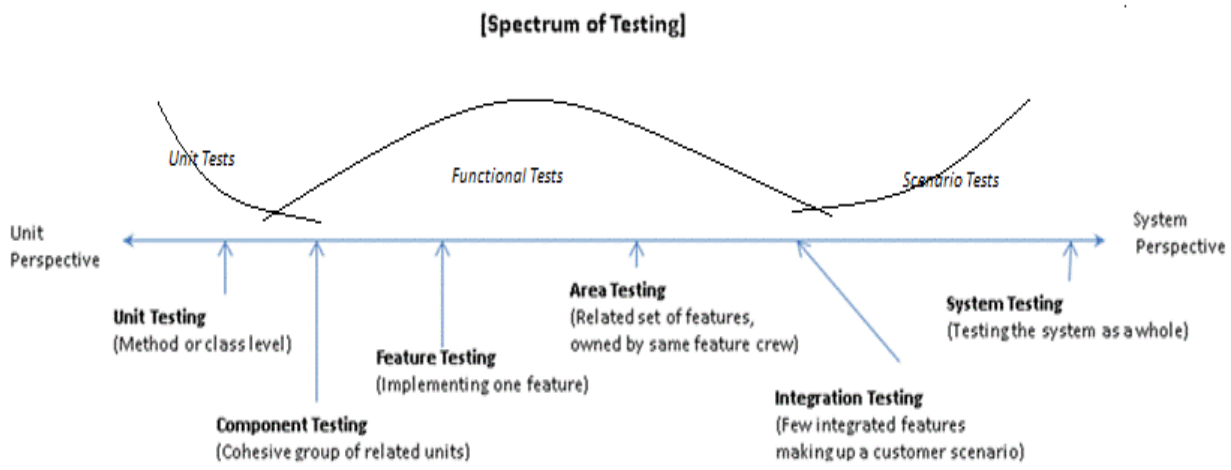
3.4 Degrading Trust for Existing Test Coverage

One of the most common problems with the lack of coverage for untold stories is that after code-complete (i.e. all stories with the implementation of features and tests for each story are complete), there is still a significant number of defects in the defect database mostly due to integration points between completed stories. This significantly degrades trust for existing test coverage as none of these defects fail existing test cases.

4. Techniques for Capturing Links among Stories

4.1 Using Test Spectrum

Test spectrum is one of the tools that the Visualization & Modeling Tools team devised and has been using to help improve test coverage. This spectrum provides a broad view on what type of testing needs to be done to verify the product. It also reminds the team that in order to have optimal coverage, they need to have a distribution of test cases across the spectrum, including touch-points among features implemented with area, integration and system level testing.



The following list describes the tests used within this spectrum,

Unit Tests: Tests that confirm a single unit of execution (e.g. a public method) behaves as expected.

Component Tests: Targets a few related units together to see if they work as expected. The majority of tests that are implemented as part of development (TDD) will fall into this category.

Feature Tests: Targets whether a product feature works as expected. I would expect most of these tests will be implemented as part of development (at least positive paths) and some edge cases, while negative paths might be delayed a few days. Feature tests can be implemented as either functional tests or as scenario tests (definitions below).

Area Tests: Targets how a series of features intersect with each other within the scope of feature crew ownership. I expect major paths are implemented as part of development and that all these tests should be done within sprint.

Integration Tests: Targets features across feature crew boundaries, making up a customer scenario. I expect new paths that are added by the feature crew should be tested within the sprint as well as all paths changed due to features implemented by feature crew.

System Tests: Targets the system as a whole, potentially new or changed paths need to be included in test planning within the sprint, and tests should be available as integrations happen. End-to-end scenarios are good examples of system level testing. Stress testing, performance testing, etc. can be considered as a cross-cutting set of test categories presented with the spectrum. From a coverage perspective, such individual test cases would fall into one of the following test categories.

4.1.1 Test Types within Spectrum

Across-the-spectrum tests can be categorized into three broader groups:

Unit Tests: Tests that confirm that a single unit of execution (e.g. a public method) behaves as expected.

Functional Tests: Functional tests should confirm that programmatic classes, objects, and methods work together to do what they're supposed to do, where possible. Functional tests are written against the public API of the product.

Scenario Tests: Tests that confirm a specific user experience. Scenario tests simulate the experience of a real end-user. Scenario tests are written against an abstraction that mimics user action. It should ideally read like a spec. It can also be written to not use UI (using the API) for better performance and robustness.

4.1.2 Test Planning Guidelines with Test Spectrum

Test spectrum is designed to remind the team that for any given feature, multiple levels of coverage are required focusing on different aspects of the implemented code. As test planning and low-level test implementation starts, keep in mind that optimal coverage could be achieved only by a combination of the various elements in the test spectrum. For the story being implemented, choose a good set that is a combination of these types – do not rely only on unit and feature tests.

As part of a test plan review checklist, it also helps to question how test coverage will be achieved at different levels. This suggests that the test plan should call out integration points with other components/features/areas that are implemented or planned. For optimal coverage, either add tests for these to the current test plan or add a task to the backlog to identify or implement these later. As other team members review tests plan/implementations, they should provide feedback aboutmissing integration pieces in the coverage.

4.2 Test Meta-Data

After test cases are designed to consider the full test spectrum, multiple levels of test cases will be created. Another dimension of meta-data is required to relate test cases to stories and provide traceability. At a minimum, each test case should have meta-data that describes: (i) components to be tested with their scenarios, (ii) test types according to the test spectrum (iii) stories that the tests are covering.

This meta-data will provide a set of attributes that can be queried to find relevant sets of test cases for each story implementation. Visual Studio 2010 provides extensible test categories and test properties to attribute test cases with proper meta-data. The Visualization & Modeling Tools test team created a set of extensions to test categories and used them to implement the Visual Studio 2010 Visualization and Modeling Feature Pack.

4.3 Using Architecture Diagrams to Relate Stories to Components

Another technique that the Visualization & Modeling Tools test team used in test planning was to include architecture diagrams as a requirement. Each test plan had one or more UML architecture diagrams that defined the context for the test plan. Using these diagrams in test plans helped relate test areas to product architecture and also question the test plan in terms of links between components/layers in the product architecture.

If there is a link between the components being tested and other components, it means there might be a code execution path that enables a story (possibly an untold one) which goes through these interacting components. This will translate as requirement to design a set of test cases in the test plan to cover the interaction.

The Visualization & Modeling Tools test team commonly used layer diagrams, UML component diagrams, and UML class diagrams to set context for the test plan and to discover potential untold stories. Visual Studio 2010 Ultimate includes these modeling diagrams.

4.4 Story Links over Architecture Diagrams

As the next step in the evolution of test techniques to cover untold stories, the Visualization & Modeling Tools test team created a set of component diagrams to improve the links between stories and the product architecture. In these diagrams, each story that has been implemented thus far overlays architectural components that implement or enable that story.

As more stories are implemented, these component diagrams encapsulate all stories overlaid on the component diagrams that represent the current architecture. If there is a link between two architectural components, it implies a link between the stories that overlay those architectural components. Each link at either level deserves an investigation on how to design test cases to coveri the interaction. If the link is at component level, the corresponding test could range from component test to integration test. If the link is at the story level, then the corresponding test could range from feature testing to system testing.

5. Planning for Untold Stories

5.1 Case Study: Closing Integration Test Gap

In some cases, a team might discover that existing test coverage is sub-optimal and needs to be improved. The following work was completed in Sprint 5 by one of the Visualization & Modeling Tools teams to improve test coverage for integration points between already implemented stories.

As part of an integration test work item in Sprint 5, QA members worked on a two-step process. First, they investigated and identified the low-level testing gaps for all areas. They used area tests only for code-coverage runs and, without integration/system level tests, figured out how much coverage gap they had with existing tests. All areas should have good low-level test coverage by only area test.

Apart from low-level testing coverage, they went over all the previously implemented stories and looked for possible integration links or scenarios across the areas that they didn't account for as part of implementing the story. Given that there were twenty-five stories that were implemented in five sprints with a significant number of interactions among them, it was a big challenge from both a story perspective and a quality perspective to account for all the enabled (intentional or unintentional) scenarios as stories come online. For example, when we implemented the Cancel story, the Update story did not exist. Cancel story enables the user to cancel a layout operation, whereas Update story enables changing various properties of nodes in the diagram. After we implemented the Update story, the functional surface that the Cancel story exposed had increased significantly. Therefore, as these two stories came online at different times, there were possibly interactions between these two stories that did not get highlighted in either of the test plans.

QA team members had brain-storming meetings to enumerate integration scenarios across components or areas as well as stories. They updated the test plans as they found these integration scenarios. It was relatively easy at this stage as the team had already started using architecture diagrams in their test plans. Going forward, all test plans will have sections for integration scenarios with enabled or previous stories as well as a section for possible feature story integration scenarios.

The team is now more successful at questioning the integration points between stories that come online and is more immune to impactful bugs that fall into the cracks between stories or components. This activity is not limited to QA; the entire team should be looking for these cracks in design discussions and story/spec reviews as they build the product daily.

6. Conclusion

Designing tests to cover integration points is a very difficult task. It gets more difficult in agile development as there are logical separations between stories or features. Designed tests usually focus on acceptance criteria and low-level coverage for the components that enable the story within sprint boundaries. It is also difficult because there is a "done" expectation at the end of each sprint, specifically, that the product should be ready to ship. It frequently adds pressure to complete the feature with very limited time left for testing.

The following steps are advised to ease issues with integration points during test design:

1. Ensure that you have a good set of test cases across the test spectrum for a given story. Do not focus only on unit tests or acceptance tests, but also carefully consider integration points for already implemented stories.
2. Mark your test cases with meta-data associated with test coverage intention, test type, and the story or feature that is related to the test. This will provide traceability and a test bed that you can query.
3. Use internal product knowledge, especially architecture diagrams, to map stories to actual code. Further, design test cases over this mapping with both links to stories and product code. This will help you to discover links among stories that would be very hard to find otherwise.
4. If you fail to identify integration points within the iteration, add high priority test coverage work items to the backlog as you discover them. Do not accumulate debt as it carries significant risk closer to the release date.

If integration points are considered carefully during agile development from testing perspective with the techniques presented in this paper, then end game stress can be more predictable and manageable.

References

Beck, K. et. al., 2001. Manifesto for Agile Software Development, *http://agilemanifesto.org/*

SIMULATING REAL-WORLD LOAD PATTERNS WHEN PLAYBACK JUST WON'T CUT IT

Wayne Roseberry

wayner@microsoft.com

Test Lead, Microsoft Corporation

ABSTRACT

Load testing is an important part of performance and reliability testing for server software. One of the goals of load testing is to determine as well as possible whether the software will stand up reliably under real-world operating conditions. One way to achieve this is to attempt to simulate in laboratory conditions the traffic and usage patterns that will happen in the real world. The challenge is to create a simulation that models the real-world behavior accurately enough that the results can be trusted as representative and expose product flaws. There are tools available, which provide web-traffic capture and playback, but these tools are lacking when it comes to many complicated client/server protocols and feature scenarios. The goal, then, is to create systems that can capture the data and traffic patterns from a production system, express them as a model, and drive the load test from that model.

This document describes one such tool built by the SharePoint team in Microsoft Office 2010. The document describes the architecture and behavior of the tool, the ways it was applied, the types of product flaws it exposed, as well as limitations of the tool and possible improvements for the future.

BIOGRAPHY

Wayne Roseberry has been in the software industry for twenty-four years, twenty of those at Microsoft Corporation, sixteen years testing software. His product experience includes Microsoft Office, The Microsoft Network (MSN), Microsoft Commercial Internet Server, Site Server, and SharePoint.

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

The newest release of SharePoint in 2010 adds the ability to use web-browser based versions of the Microsoft Office client applications for content viewing and authoring. SharePoint's end user interfaces support web browser, rich GUI client, as well as SOAP and REST based web service programmatic interfaces. The technology stack is built on Internet Information Server, ASP .NET and SQL Server. Further, the Microsoft Office clients, from version XP through 2010 support integration with SharePoint for file storage, workflow integration, and social collaboration. There is a rich API for server side applications, allowing a customer or third parties to extend behaviors to match their business needs.

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 PREVIOUS TESTING CHALLENGES

Simulated real world load testing of SharePoint is extremely difficult. There is a great deal of complexity and size in the number of different customer scenarios. However, it is likewise difficult to apply a quality bar against tests without mapping expectations against real world patterns. Here are several of the issues that come up:

1.2.1 ABNORMAL VS. TYPICAL USAGE PATTERN DISTINCTIONS ARE HARD TO DETERMINE

It is easy to create load tests that will push the system to its limits and create performance or reliability problems. The difficulty comes in triaging those problems to determine whether they merit a fix in product code. Fixes in SharePoint frequently involve tuning SQL queries, the performance and behavior of which are highly impacted by data and usage patterns. The same fix that works well in one case may be a very bad choice in another case, so it is very important to be aware of the most common usage patterns when selecting performance fixes. This motivates the test team to seek out usage and data patterns that are already validated by production systems.

This becomes difficult with SharePoint because the functionality is broad, and the use cases and scenarios highly varied. In a similar manner, the data sets are frequently large. It is not uncommon for a customer to have multiple terabytes of data in production.

1.2.2 DYNAMIC STATE MAKES SIMPLE PLAYBACK TESTING NON-EFFECTIVE

Capture and playback systems will record requests on a communication channel and then play those requests later on a system under test, usually with a copy of the data that existed on a system at the time

of capture. This works effectively when the communication is purely stateless and predictable. An example would be a web page that always sends the same reply when the client issues a specific request.

The problem occurs when a web page generates dynamic data, or is highly dependent upon a complex state sequence before the request occurs. This forces the client to send requests that preserve whatever the server sent previously, or to reply at a moment when the state is still consistent. In this case, the recorded request is only valid at the moment it was captured. Playback later will be invalid because the request no longer contains the correct information, or did not happen at the moment that the server could process it correctly.

SharePoint has many features that manifest exactly this behavior. For example, when a user adds a new document to the system, that document is given a unique and random identifier that is generated by the server dynamically at runtime. Later pages that perform operations on that file, such as checking it out for edits, viewing or editing properties of the document or launching a workflow against it, pass that unique identifier in the HTTP request to the server. If that identifier does not match one for an existing document, the operation will fail. A generic capture and playback tool cannot anticipate ahead of time what the identifier will be for newly added documents; hence, the test code in this case needs to be more specialized. It needs to anticipate the identifier and remember it for use in later requests.

A similar condition occurs in complex state sequences. For example, SharePoint has a collaboration workflow sequence that can enforce a requirement that users check out a file before editing and then check in that same file before allowing others to edit it. If a user attempts check out on a document already checked out the operation will fail, and similarly if a user attempts check in on a document that is already checked in the operation will fail. The time gap between check out and check in with real world systems is unpredictable. A user may complete the sequence in just a few minutes, or may leave a document checked out or in for days or months. A capture tool, then, runs the risk of capturing a document in the middle of the sequence, and then on repeated replay invalidates the sequence. The mitigation is not to use generic tools, but again to write test automation that is aware of the state of the system and apply the tests properly according to the rules of the system (or at least forcing the sequences you desire to test rather than having them break because the capture system is not flexible enough).

1.2.3 DATA SAMPLES ARE CRITICAL TO PERFORMANCE AND RELIABILITY

SharePoint data sets are large and complex. There are many databases in a SharePoint deployment, and many different types of objects that can grow in size per object and in quantity of those objects. For example, a document library may contain files that are very large or very small, and likewise can contain just a few documents, or perhaps millions of documents. These two different variables for document libraries combine together to make the actual performance characteristics more complex than if it were just a matter of size of file or number of files. Just as usage patterns will push the database engine to extremes that may not be representative of your customer needs, so will the data sets. Different data set characteristics, and the frequency at which end users access that data can make the code perform in very different ways. For example, there are several places within the SharePoint stack, which cache data to save on back end round trips or SQL load. As the number of distinct items requested grows larger, so does the size of the cache. As the cache gets larger it becomes necessary to trim the cache, thus slowing down processing, but also increasing round trips to the back end to re-populated items removed from the cache. Further, the larger the set of popular items in the system, the more likely it will be that there is a cache miss. This means that in order to maintain realistic cache hit, miss and maintenance patterns, the test should be designed to imitate both relative popularity and total quantity of different resources requested. When the data sets and access patterns in tests push the system to extremes beyond what the customer is expected to require the resulting defects are difficult to prioritize. The fix may optimize for the narrow case at the risk of regressions in the more common case.

This motivates the test team to look for real-world production system data that has a corresponding real-world traffic pattern to match. This is difficult, as the production data sets are often larger than the test

team resources can store. Another problem is that production data usually falls under privacy, legal and intellectual property restrictions. Such restrictions may even apply within the same company. For example, the personal site data for employees in a company may not be available outside the IT facility where it is deployed.

1.2.4 INVESTIGATION IN PRODUCTION IS EXPENSIVE, SLOW AND TOO FAR DOWN THE PIPE

The SharePoint team uses internal production systems for long periods to discover issues in the product. While this technique yields a large volume of high value flaw discovery it is an expensive means to find bugs. Downtime on a production system means work stoppage for everybody using the system. Production systems run under complicated conditions and the patterns of usage are unpredictable. Many debugging and diagnostic procedures are unavailable. The code running on the production system is also usually multiple weeks and eventually months behind the code that is in development. All of this motivates a desire to move bug discovery upstream in laboratory tests that simulate the production environment.

1.3 WHAT WE PLANNED TO ACHIEVE

Based on all of these issues, we set out the following goals:

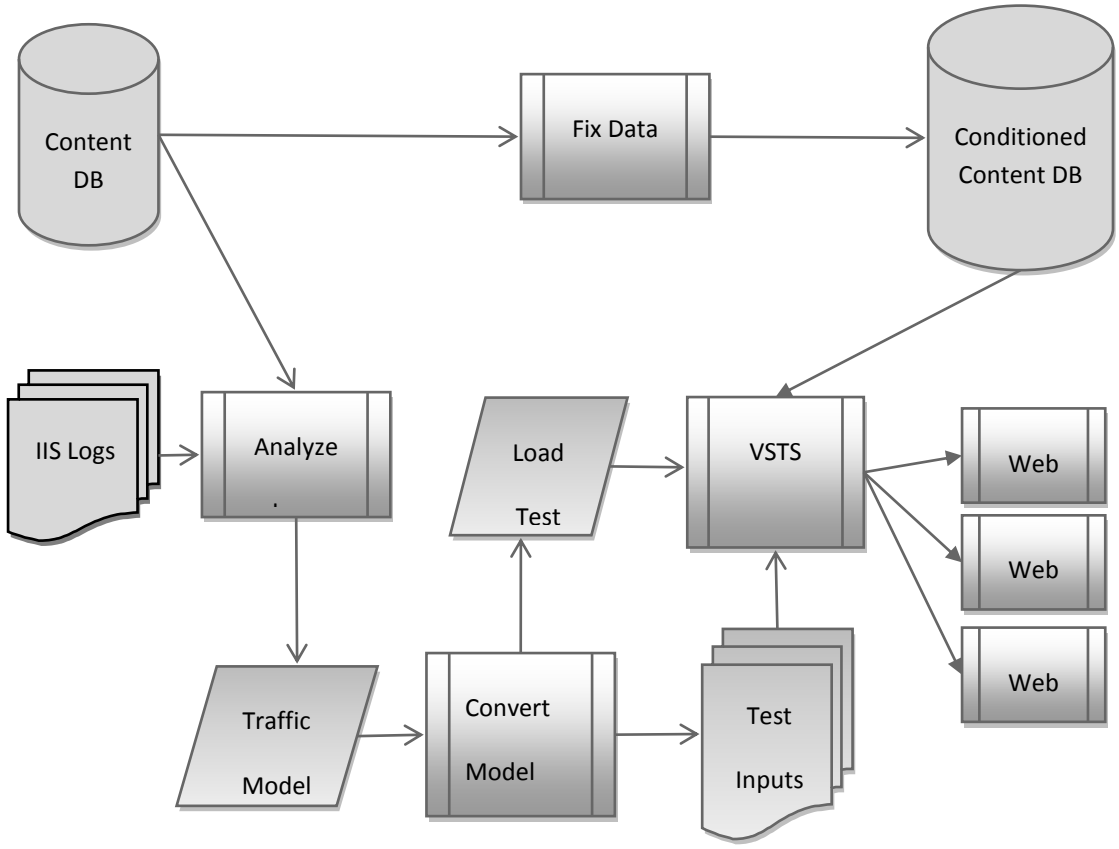
1. Run tests that predict the performance and reliability flaws that manifest on our production systems before deployment
2. Find usage patterns from the real-world samples that manifest bugs that are otherwise hard to discover
3. Simulate real-world traffic patterns taken from existing customer and internal systems to help calibrate our awareness of the importance of different bug fixes, performance goals and objectives
4. Create a regression suite that can be used for non-production problem investigation and code fix validation before checking source code into the project
5. Create a test lab environment that we can use for inventing test methodologies for investigation and diagnosis
6. Re-use the solution to help customers with their own capacity planning and performance investigation activities

2 SYSTEM DESCRIPTION

We built a load simulation tool that was designed to satisfy these needs. This tool was first used in SharePoint 2010. It is designed to sample existing traffic data (taken from Microsoft Internet Information Server logs on a customer machine) and content (taken from existing SharePoint database) and generate the data files necessary to tell a load test what to do. It uses Microsoft Visual Studio Test System (VSTS) to generate, load, and execute the tests. By basing the load component on a shipping product, we achieved portability of the test to any environment.

2.1 TOOL ARCHITECTURE OVERVIEW

There are several parts to the system:



1. Analyze the IIS logs to establish request types and traffic patterns (Traffic Model)
2. Build a model that describes the distribution of request types and resources
3. Convert the model into the input data for the load test
4. Copy the Content DB to the test system
5. Condition the content to make it test ready ("Fix Data" in the diagram)
6. A set of web tests capable of executing different types of operations against the system under test

2.2 ANALYZE THE IIS LOGS TO ESTABLISH REQUEST TYPES AND TRAFFIC PATTERNS

2.2.1 REQUEST CLASSIFICATION

The Traffic Analysis component processes the IIS logs a line at a time and looks for information in the request URL, request type and User-Agent to determine what type of request the user executed. This is important because not all requests can be accurately executed as a generic GET or POST action against the same resource. Specialized test code is required to simulate accurately the scenario, and identifying the type of request is the step that establishes which test code to use to process the request.

The traffic analysis component used a set of pattern matching rules to narrow down the request type. Here is an example of some of those rules:

Test Name	Output File	Object Type	IIS Criteria
ECM_GetStaticFileInDocLib	file.csv	File	csMethod = 'GET' and csUriStem not like '%/personal/%' and csUriStem not like '%/_layouts/%' and substring(csUriStem, len(csUriStem)-charindex('.', reverse(csUriStem)) + 2, charindex('.', reverse(csUriStem))) in ('jpg', 'gif', 'css', 'js', 'pgn', 'axd', 'bmp', 'ico')
WSS_ViewListRSSFeed	list.csv	ListFeed	csMethod = 'GET' and csUriStem like '%/listfeed.aspx'
MOSS_MySite_ViewProfilePage_ByEveryone	accountname.csv	MySitePublic	csMethod = 'GET' and csUriStem like '%/Person.aspx' and csUriQuery like 'accountname%'
MOSS_MySite_SharedDocLib_ViewAllItemsInFolder	doclib.csv	FolderView	csMethod = 'GET' and csUserAgent like 'Mozilla%' and csUriStem like '%/Shared+Documents/forms/allitems.aspx' and csUriQuery like 'RootFolder=%'

In the above example, “IIS Criteria” is a query used to match content found on the HTTP request entry in the server’s log file. If the query matches, then the request is assigned to the web test defined in the “Test Name” column. The “Object Type” column indicates the type of action or resource the end user was accessing in the request. The “Output File” column indicates which test data file to store test parameters in for this request.

2.2.2 REQUEST AND USER DISTRIBUTION

The analysis tool also tracks which users and which resources are accessed how often on the production system. This step has to happen after request classification because different parts of the request data may mean different things to different types of tests. For example, some tests may simply require the full URL to determine what to do. Other tests may ignore parts of the URL and look at only a specific part of the path or just the filename. Other tests may require extracting part of the QueryString (the part of a URL request following the question mark) to further isolate specific operations or data locations.

2.2.3 USER PERMISSION ANALYSIS

Another part of the analysis is to traverse the content database and determine which users have what permissions levels inside the content. SharePoint allows per user and per group permissions at the entire site, specific site, specific library and in some cases file level. Operational costs vary a great deal per permission level, so modeling the permissions accurately is necessary to avoid creating unrealistic load patterns.

2.3 TRANSLATE THE MODEL INTO TEST DATA

The output of the analysis phase is a file expressing a model of the percentage distribution of the different operations in the IIS log. The percentage of users and resources accessed are likewise recorded in the model on a per operation basis. The following is an example:

testname	Distribution
Ecm_GetStaticResourcesInDocLib	16.148
Ecm_ViewSiteHomepage	3.621
Wss_ViewSiteHomepage	5.775
Wss_ViewCustomDocLibView	0.921
Wss_ViewListAllItemsPage	0.480

Wss_ViewDocLibAllItemsPage	1.042
Wss_ViewAllSiteContentPage	0.083
Wss_ViewListRSSFeed	1.214
Wss_ViewStandardRSSFeed	0.081
Wss_Download_SaveTargetAs	0.888
Wss_Download_OpenInReadOnlyMode	48.400
Wss_Download_OpenInEditMode	21.347

In the above example, “testname” refers to the name of a web test in Visual Studio (described later in this document), and “distribution” refers to the percentage of time that test should be given during the run. .

The next step is to convert this model into a format that can be consumed by the web tests. There are two parts of this:

1. A Visual Studio Load Test file
   Visual Studio load test files describe the behaviors of a test run; which tests to execute, what percentage weighting to give, how to model the load pattern on the test harness, etc. The conversion from model to load test is straightforward, the frequency of the operations identified in the model are mapped directly to the percentage weighting of the tests that correspond to those operations.
2. Test data sources
   Visual Studio allows binding between a test and a data source to describe inputs to the test. In this case, the inputs map to the resources and users associated with the operation during the analysis phase. The data sources are constructed such that more popular resources and users that are more active are used in the test run than less popular resources or active users in proportion to what was observed in the real-world sample. In this instance, the data sources were simple text files in CSV (Comma Separated Values) format.

At this point, Visual Studio has enough information about the test model to execute the tests, users, and resource access patterns in the same percentage as was observed in the production system. All that is required from this point are an available site prepared with a copy of the data and a set of web tests able to execute the operations.

2.4 WEB TESTS

Visual Studio Test System uses two types of objects to do a test run. The outer most object is the load test, which is really just a container for mixing test code together in a run different test inputs, percentages and run time conditions. The inner most object is a web test, which actually executes the operation being tested. There are simple web tests, which just play back URL requests and allow test development without any coding, and there are coded web tests, which allow you to introduce complex test behaviors via compiled code that go beyond the capabilities of the simple web tests. The web tests are really where the tests happen. The load test is just a container to coordinate different web tests in a mix.

The web tests are actually an independent part of the system. The SharePoint team was developing them already in order to do isolated load tests against single components and operations, as well as simple mixing. Some examples of web tests include; uploading documents, rendering a spreadsheet online, initiating a workflow on an item in a list, filling out a custom form and submitting it to a list or simulating synching of RSS (web feeds that publish frequently updated works) feeds to a client application. Several dozen web tests were developed that represented approximate 99.95% of the request classifications that we had seen from traffic in our internal 2007 deployments.

The important part of this system was to author the web tests in such a way that their behavior was driven by input files built from the traffic model. This allowed the tests to adapt to the traffic model rather than impose their own patterns independent of the model.

While not directly part of the real-world modeling system, the bulk of the test development time was spent building and adjusting the web tests. As noted later in this document, feature and operational coverage are a significant part of the value of the pattern. The value of the test run diminishes substantially for every operation, or permutation on operational behavior not represented with a test. Making this test development even more difficult is the fact that SharePoint is built by approximately twenty different teams within the Office organization. Coordinating the development, delivery, analysis and maintenance of these components took a great deal of effort.

2.5 CONTENT AND DATABASE COPY AND PREPARATION

The original content database copied to the test system must be modified before test execution. A copy of the actual data is used so that when the web tests execute they request resources that actually exist, but the data in its original form is not ready for test.

User permissions must be changed on the database before the system can be tested. The users from the production system are real users that live in an actual network. Their user identities are mapped to security properties that determine their permission within the system. However, the password and credentials of those users are not available to the test system. This means that the real users must be replaced with artificial users coming from an Active Directory inside a domain that is under control of the test system.

This is where the permission modeling comes in. The test accounts are mapped to real users from the model, and then the test accounts are mapped to groups. Those groups are then given permissions inside the copy of the content database that mimics the distribution of permissions of the real users in the original production system. This condition allows the web test to log into the test system with the artificial test user credentials and have the same permission levels of the users from the production system.

2.6 PERFORMANCE AND RELIABILITY MONITORING

We included an in-house performance and reliability monitoring solution during the runs in addition to Visual Studio's own data collection. This monitoring solution was the same one we were using on our production system, allowing us to gather the same statistics during production and lab runs and compare results.

The monitoring tool executes a "ping" in the form of an HTTP request against a known set of URL's on the system under test. The result of this ping was used to measure time to last byte in the request, errors, and timeouts. These were aggregated to collect data on page latency percentages (time to last byte at 25%, 75% and 95% of the sample), availability (percentage of time the service was up versus down) and failure rate (percentage of requests returning an error or timing out).

3 APPLICATION

We used this load simulation system, or parts of it multiple times during the SharePoint 2010 development cycle. In addition to modeling real production systems, it also afforded us the opportunity to re-sample the traffic pattern and update the model to accommodate changing usage patterns as new features are brought into production.

Below are some examples of real-world deployments we modeled.

3.1 OFFICE DIVISION PORTAL SIMULATED LOAD TEST

The Microsoft Office team uses a SharePoint portal for its own divisional point of access to a variety of information and tools. Office has approximately 7,000 users in the organization, and the site is used to

communicate information about Office to the rest of the company, coordinate project development and communication about new versions of Office to the Office team, coordinate communication and collaboration with partner teams within Microsoft and serves as a place for teams within Office to coordinate their individual features and projects. The site has archives of data and documents regarding previous projects, which are used for sustained engineering and support. It also has a single library that is used to house all of the feature requirements, design documents, tests specifications, and general development collateral for an entire release of Office. The site receives about 10,000 unique visitors a week and about 7500 unique visitors a day. In addition to servicing collaboration needs of the division, the site also has personalized site and user profile features that would normally be shared across an enterprise – this last aspect of the site is done to give the Office team an opportunity to put the software into production and experiment with feature behavior before the entire company adopts the features.

In releases previous to this Office 2010, deploying this pre-production system took considerable time. A substantial part of this effort was addressing performance and stability issues. To mitigate this in the 2010 release, one of the goals of the performance testing effort for Office 2010 was to test the Office portal site ahead of the production deployment, allowing an opportunity to identify major performance issues early and fix before going into production.

This was the first system that was sampled, modeled, and simulated in the load test system. Most of the necessary web tests for the remaining samples were developed to satisfy the Office portal site.

3.2 MICROSOFT IT'S HOSTED COLLABORATION PORTAL SIMULATION, AND DEALING WITH PRIVACY ISSUES

The next major sites we sampled were two team collaboration sites hosted by Microsoft's own IT group. These two portals are a location where any team within Microsoft can self-deploy their own site. The sites receive approximate 80,000 unique visitors per day.

The usage patterns between the IT hosted solution and the Office divisional portal are similar, but differ in important ways:

1. The Office site represents collaboration of a single division, which means that the patterns of usage vary as the Office team moves through its project schedule. This means that there are times of the year where document authoring was far more common, and then long spans where very little document authoring happened at all.
2. The IT hosted site represents the collaboration of hundreds of teams, all at different stages in their project cycles and with different classes of business needs. This spreads the usage pattern more evenly across time, meaning you do not see as much seasonal variation in usage.
3. The Office team is worldwide, but predominantly resides in the Redmond, WA campus. This means that while there are usage humps during business hours in Asia and Europe, they are not as pronounced as the ones in Washington.
4. The IT hosted solution represents a far larger distribution of employees outside the Redmond, WA area, which means a larger usage hump as the business hours move across the globe.
5. The Office site user behavior has a higher usage rate (Office users on average access the Office site more often than users of the IT solution did), but the volume of users accessing the IT site on a daily basis push the IT system to about twice the throughput rate of the Office team solution (Office site at 8000 users with 155 Requests Per Second (RPS) peak, IT hosted site with 80,000 users at 304 peak RPS).
6. No other team on the IT hosted solution has created a single document library as large as the Office specification library. The activity on the IT hosted solution tends toward smaller lists and libraries.
7. The data on the IT hosted solution falls under more sensitive privacy and legal restrictions.

For the most part, the IT hosted solution represented an opportunity to observe traffic patterns that we felt represented a more typical case that tracked more closely to what many customers would experience. It

also represented a chance to run our software on a production system with more rigorous business demands and hence less tolerance for problems and failure than with the Office product group.

From an engineering perspective, however, the site represented a more interesting challenge. The data set overall on the IT hosted solution was much larger than we had room to host in the test laboratory. Further, the legal and privacy restrictions prevented us from copying the entire content database out of the production facility and into our test laboratory. We were only allowed to copy an approved subset of the data to our test laboratory and work with it from there.

This forced us to modify the load simulation process slightly. We still wanted to represent the volume and percentage mix of users and resources based on a real system, but we could not use all the content data to do it. Therefore, we added an incremental step. We took the model described by a sample from the full system, and then mapped the resources described in that model to just use the subset of the data we were able to copy to our lab. This allowed us to describe the traffic pattern we wanted and satisfy the legal, privacy and size limitations of using the production data.

This created a problem. The smaller data set meant we were not exactly modeling the scale characteristics of the system. However, we did have larger scale testing results on artificial load and data sets, just on single variables at a time. For example, we had tests that measured how the number of files in a library affected performance, and another that measured how the number of sites inside a website-affected performance, and likewise for other variables, but not how all those variables affected each other in combination. We used these results on independent variables and extrapolated back to our real world test run to assess how we expected the real world performance to behave as it went to larger scale levels. We also relied on aspects of the system design that we knew could scale independently. For example, the total number of sites in a SharePoint system can grow infinitely so long as you continue adding databases and database servers to handle the data set, so we knew that as long as we established the scale potential of a single database server we had enough information to infer the rest of the scale requirements without building an exact replica of the real world deployment. These extrapolations were obviously compromises, but were ones we believed came with manageable risks.

3.3 MICROSOFT'S IT HOSTED PERSONAL SITE SIMULATION

SharePoint has a feature that allows hosting of personal sites for all employees within an Enterprise. The personal site has a profile page for every person with data about them regarding office location, location in the organization hierarchy, expertise or any other piece of information the company chooses to track. Likewise, each user can have a site for keeping personal documents, documents they share with others, lists to track activity and collaborate with others.

Microsoft IT hosts a personal site portal for all employees with the company. There are approximately 150k employees hosted on this portal. The site receives approximate 71,000 unique users per day generating an average of 93 RPS. By contrast, the MS IT hosted collaboration portal, at 80,000 unique users and 304 RPS generates approximately 3 times the load per user.

The traffic on the IT site is dominated by client synchronization requests for RSS feeds, and in the Office 2010 release activity feed and social network updates from Outlook. Part of the reason for the high volume of RSS feeds is because Microsoft IT has configured a default synchronization feed to every user's client to their shared document library.

The personal site data has a similar size and security policy problem as the hosted collaboration solution, with the end user privacy issues being even more restricted. The test-engineering problem was very similar, we had to map real users to artificial users in order to obscure identity, and we were only permitted a very specific subset of the user data.

An even more difficult problem arose regarding personal site visits. When a user visits a portal site, their identity is detected and they are shown a profile page. This page renders differently when a user is viewing their own page versus when they are viewing someone else's. However, since we were using

artificial test accounts to run the tests we were not able to visit a person's own profile page. To solve this problem, we gave the artificial accounts the same permission as the site owner to the personal site of each real user.

3.4 CAPACITY PLANNING TOOLS AND DOCUMENTATION

Customers that purchase and deploy SharePoint need to purchase hardware and plan their configuration. They require a set of documentation and tools that help them determine what hardware will meet their needs, and how to properly configure and tune that hardware once deployed. Such a set of tools were created and published with SharePoint. The test tools described in this document were a critical component in preparation of that toolkit and documentation.

3.4.1 CAPACITY PLANNING REPORTS

The capacity planning reports and documentation for SharePoint 2010 used the load test solution described in this document. These reports are available on the web at the following locations:

All capacity planning case studies can be found here:

http://technet.microsoft.com/en-us/library/cc261716.aspx

Site From This Document	Report name on website
Office Product Group Portal	Departmental Collaboration
Microsoft IT Hosted Collaboration Portal	Intranet Collaboration
Microsoft IT Hosted Personal Site Portal	Social

Further, load pattern and resource usage analysis reports of the actual production system are included in the same website. Basing our capacity planning processes on loads modeled from real production systems allowed us to show impacts of scalability and load changes and then allow customers to compare them to the real system. This gives the customer some visibility into the difference between the test run data (which is somewhat artificial) and the real experience, thus we hope making it easier to calibrate the test data and make their own planning easier.

3.4.2 LOAD TEST KIT & EXTERNAL CUSTOMER EXPERIENCE

Before release, the testing tool was packaged as a customer consumable load testing kit. The idea being that the customer could base hardware purchases and configuration choices on real traffic patterns from their existing deployments rather than making blind guesses. We had an opportunity to work with one of the customers on the early adoption program with this test kit.

An interesting example of an issue that came up during this customer test is one similar to something we experienced internally. The customer had 3 terabytes of data in their production system, but was only able to use about 500 gigabytes in the test run due to cost constraints. The log file that the test kit analyzed, however, contained requests for the entire production data set. The solution to this problem was similar to our own; requests for missing data were mapped to test data that existed in the laboratory.

Other challenges came up that centered mostly on education about the test framework, familiarity with performance planning in general and extending the test suite to cover additional behaviors. Load testing and capacity planning are complex problems, and extending what you have learned to do before release out to your own customers is a difficult task.

In general, though, we found the tool an effective aid to help the customer to plan their deployment and make hardware decisions.

4 DEFECT FIX AND FIND RATES

Simulation Runs

The simulation runs were considered a very successful project, resulting in a high defect find rate. In the table below are the bug numbers found by two of the simulation tests mentioned in this document (the Office team portal and the IT hosted collaboration portal). Also included are defect resolution percentages, which refer to whether the defect was ultimately fixed or not. The resolution legend is as follows:

By Design: The behavior reported is intentional and not a bug

Dupe: This problem has been reported already

Ext.: This behavior is due to a defect in an external component

Fixed: The defects that were fixed

Not Repro: Attempts to reproduce the problem failed

Postponed: The defect will be reconsidered for fixing in a later release

Won't Fix: The defect will not be fixed

There are two rows of bug totals, one for performance related defects, and "Other" which refers to not-performance defects found as a side-effect of the performance testing process such as pages that failed to render, errors found in product logs during execution, failed data set upgrades, or other various functionality related failures.

Simulated Test Run Defect Find and Resolution Rates

	By Design		Dupe		Ext.		Fixed		Not Repro		Postponed		Won't Fix		Total #
	#	%	#	%	#	%	#	%	#	%	#	%	#	%	
Performance	25	8%	84	27%	11	4%	88	28%	43	14%	3	1%	55	18%	309
Other	13	10%	30	23%	5	4%	57	43%	20	15%		0%	8	6%	133
Grand Total	**38**	**9%**	**114**	**26%**	**16**	**4%**	**145**	**33%**	**63**	**14%**	**3**	**1%**	**63**	**14%**	**442**

It is interesting to compare the resolution rates to similar bugs found by different test methodology. The following chart shows resolution of all performance defects, filed against the same teams as in the chart above, during the same time period.

Performance Defect Resolution Rate Overall (from a total set of 19269 reported defects)

	By Design	Duplicate	External	Fixed	Not Repro	Postponed	Won't Fix
%	4.21%	14.39%	1.90%	47.49%	8.31%	3.60%	19.83%

There are some substantial differences in this data. The simulated runs found about 1.6% of the total performance defects (309/19269) during the same time period. This is partly due to the complexity of the simulated runs, but mostly due to staffing differences. The simulated real-world runs were executed and defects filed by 27 (11 per) testers, whereas 1521 (12 per) testers filed the remainder of performance defects, so the bug yield per person was comparable for the same class of defects.

A larger difference can be noted in the resolution rates. For performance bugs overall, the fix rate was 17% higher than on the simulated runs, picking up substantial differences in Dupe (-13%) and Not Repro (-6%). The reasons for these differences are not well understood at this point, but likely deserve further analysis.

5 LIMITATIONS & OPPORTUNITIES FOR FURTHER DEVELOPMENT

While the solution provided much value, there are still opportunities for improvement. Some of those are described here.

5.1 COVERAGE LIMITATIONS

The impact of accurate modeling is very rapidly overwhelmed by the limitation of missing coverage. The system is only capable of testing requests for which there is an existing web test written. For simple requests, this is easily handled by playing back the exact request, but for more complicated requests the missing edge conditions and user scenarios have a large impact on the performance characteristics of the system.

What we have observed is that single actions have a dramatic effect on performance. The software overall may be well within performance goals, but a customer may still experience bad performance when less frequent but very expensive events cause momentarily high latencies or outages. The causes and classifications of these events are varied (bad database queries, synchronous events blocking on serialized resources, memory leaks, etc.)The challenge in their discovery is in creating tests that will reproduce such conditions.

There are a couple of examples in areas where missing coverage contributed to defects that escaped testing and didn't show up until production:

> **Co-Authoring**: SharePoint 2010 supports a file save, open and editing protocol that enables multiple users to work on a document at the same time from different clients on different machines. The Office 2010 client applications utilize this protocol. The protocol is very rich and complex, and correct use of the protocol requires full understanding of the file formats being edited. This complexity made it difficult to deliver as broad a set of web tests as were necessary to identify performance and reliability problems. Several critical flaws escaped runs in the test environment and manifested the production MS IT-hosted collaboration portal before they were discovered by real end users and eliminated by the product team.
>
> **Profile Synchronization**: The personalization features in SharePoint 2010 allow synchronization of user data from external data stores (Active Directory, LDAP, SQL databases, SAP, etc.) to a SharePoint profile store that is used for page rendering, local business application logic, and other scenarios. This synchronization activity impacts the same resources that are used to deliver user data. The desired test was to run this synchronization in the lab concurrent with simulated for end user traffic on the personalized site portal. Schedule and costing issues delayed development of this specific test, which again meant that several issues were not discovered until Microsoft IT's hosted portal deployed the full system.

Test automation gaps are nothing new or unique, so the point is not to confess that we experienced the same thing. The point is to raise the question of priority of more coverage over more accurate modeling. It seems reasonable to assume that accuracy in load modeling need only get to a "close enough" level of precision. By contrast, establishing "good enough" on coverage requires a great deal more effort, and seems like missing it comes at a higher cost. Perhaps the question merits more investigation, but when it

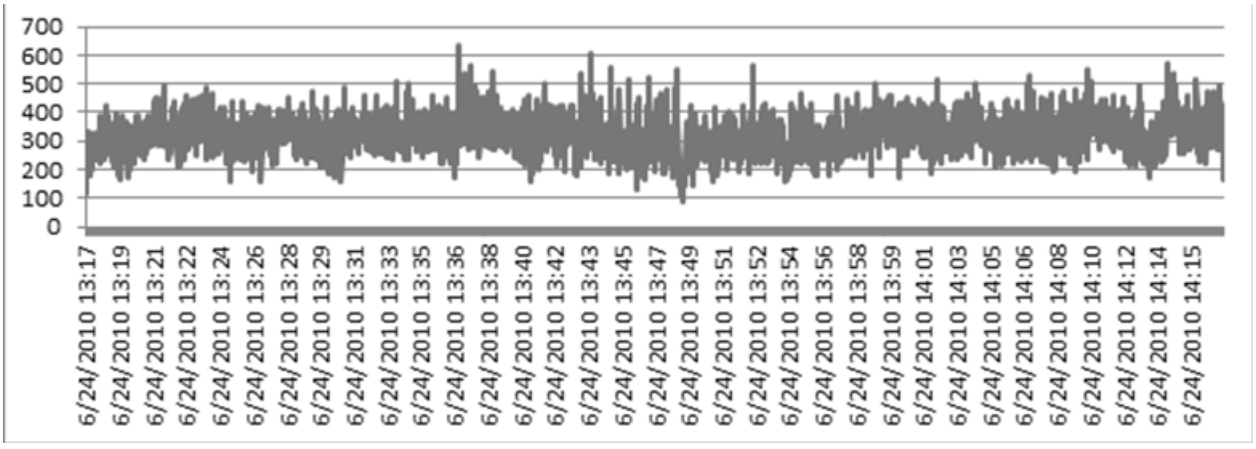
comes to choosing where to spend the money, it seems best to put most of the effort into achieving as much coverage as possible and make trades in the accuracy of the load simulation distributions once they are close enough.

5.2 TRAFFIC PATTERN FLATTENING

The traffic analysis phase of the tool in this document models frequencies as a percentage. If there were 100 requests sampled in an hour, and an operation occurred ten times, then the model will set that operation as 10% of the load.

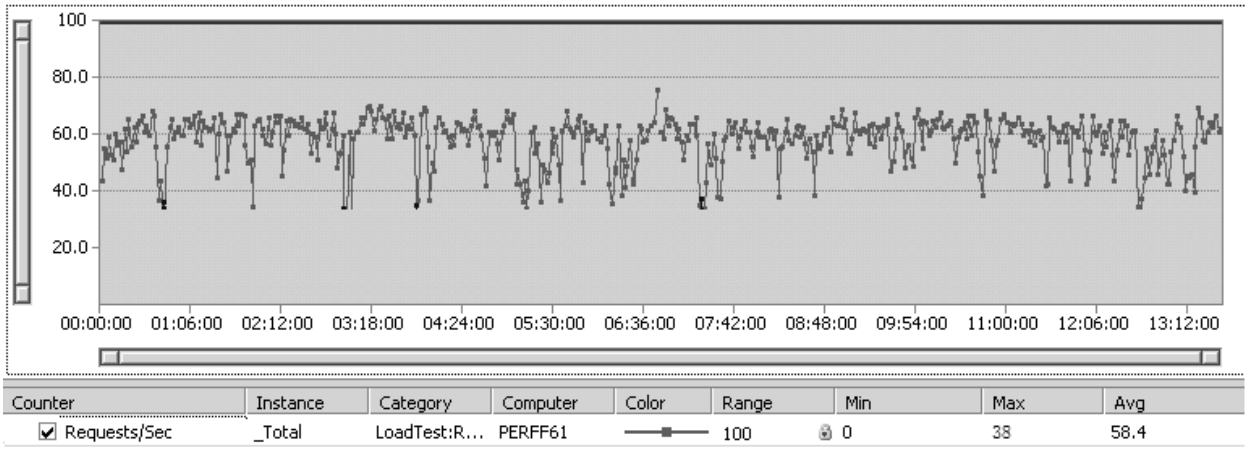
This approach does a pretty good job of modeling typical load and resource usage. If the RPS volume is taken up close to the system maximum it does reasonable job of predicting system resource utilization (memory, CPU, disk IO) for that load.

What it fails to represent are bursts and spikes in requests that happen over very short intervals. A typical graph of RPS for a production system looks like the following (taken over a one hour basis)



As can be observed from the graph above RPS vary widely over time, with as much as a six times difference between the high and low points. Average values per hour, per minute and per second have increasingly larger variances.

By contrast, the RPS pattern from the simulation is relatively flat and constant



The line in the above graph center stays much closer to the average of 58.4 requests per second. The maximum value of 58.4 requests per second is only one and a half times higher than the minimum value of 38 requests per second.

The reason for this is the following:

1. The load test examines traffic one request at a time and not in groups, hence requests don't cluster together in the simulation the way they do in the real world.
2. The load test model flattens all variations in the pattern to a percentage, which then spreads request rate evenly over time.

The concern is that the simulations are missing spikes in resource utilization, so while the general resource consumption is close to production need, the momentary spikes that cause problems are not represented. What is not understood yet is the significance in this gap in the simulation.

It could be that modeling these variations in usage as clusters and spikes may or may not yield any substantial discovery of performance defects via the test simulation. That said, further investigation of ways to represent spikes more accurately in the model might turn out to be worth the investment.

5.3 IMPROVED COST OF EXECUTION

Something not mentioned in this document so far has been the cost of creating the test environment. Every attempt was made to create a deployment that was similar (if not identical) to the configuration used in the production systems. Even with simplifications of the configuration, the resulting test environment was very expensive to create, deploy new code to, apply settings, load with data, and generally get ready for execution.

In the previous release, test runs on environments of similar complexity took weeks to build and prepare. A great deal of time was lost discovering and attempting to mitigate product bugs that prevented deployment or that blocked testing. For example, there were numerous times during the schedule when data set upgrade defects in the product would manifest on the data sets used in the performance tests, causing the upgrade of the data to abort, hence delaying the entire test until a fix was delivered for the upgrade failure. However, even with those issues resolved, a substantial part of the cost was in the complexity of setting up and configuring the test environment.

Deployment went from taking weeks down to approximately four hours. We achieved this by automating the process of copying data, installing the product, and setting up the basic configuration. The difference in time was critical, because it permitted us to begin a simulated test run within one day of a build either from the main development pipeline or from private releases from a single developer. This allowed us eventually to use the simulated runs more often for regression testing, investigating issues, bug reproduction, and eventually preparing capacity planning documentation for customer consumption.

Regarding priorities, there is a temptation to invest only in the test code and leave steps like installation, configuration, and environment preparation to test personnel or lab staff. Our experience was the delays and costs made such tests prohibitive and of marginal value. The recommendation, then, is to invest substantially in efforts to automate quick and rapid deployment of test ready environments. Again, it is likely this will have more return on value than pursuing high accuracy in the test simulation.

5.4 LARGE RETURN FROM MONITORING INVESTMENTS

The SharePoint team invested in a number of monitoring solutions, some built into the product, and others built as external tools. These monitoring systems gave us data on availability, request failure rates, percentile spreads on page latency, system crashes, and a variety of other metrics on system reliability and performance. We also built a number of tools that permitted deep, rapid investigation into source and root cause of failures.

We deployed these monitoring solutions on all of our internal production systems. We used the same monitoring solutions on our test runs. This allowed us to compare our metrics from a test run to production systems. It also gave us the same toolset for investigating root cause in lab runs as in production, which made reproduction and regression testing easier.

Even more important, though, was the efficiency at finding and filing bug failures. Before tool creation, filing a single performance bug would take days of mining data, looking at log entries and analyzing results. This often resulted in a bug that development was unable to diagnose. Improved monitoring and analysis tools dropped the costs from days to minutes and increased the bug discovery quantity a hundred fold.

This set of monitoring tools was critical to interpreting the results of these simulated load tests. The complexity of the run is close to the complexity of production systems, and the volume of requests to the system made manual investigation completely untenable. Like some of the other investments mentioned in this document, the monitoring and analysis component investment pays off more rapidly than accuracy in modeling.

6 CONCLUSION

Simulating real-world patterns is a challenging, but necessary part of performance and reliability testing for a server product. Simple playback and record technologies don't adequately capture the full complexity of the scenario and often are unable to simulate dynamic run-time features. This necessitates creation of specialized load tests that are aware of the product itself and which can handle the simulation appropriately. What we discovered is that while building such tools based on real world traffic samplings is difficult, it is definitely possible, and the return on investment is high. It is possible to achieve higher defect find rates and better confidence in triage priorities of the defects discovered. Further, such tools can be re-purposed effectively to accommodate different customer usage patterns and different engineering needs that go beyond just getting performance measurements.

We also discovered that this high return on investment could be achieved without establishing a perfect simulation model. There is a substantial difference between "good enough" to be worth the effort and achieving absolute accuracy. We discovered that we had high return on investment from supporting aspects of the solution, such as creating test environments more quickly and easily than doing so manually.

Several interesting questions remain after this exercise. We still do not know how much return on investment we will get on defect discovery costs and rates through more accurate simulations than we already have. We also have not thoroughly analyzed the qualitative difference between bugs found via real-world simulation versus those found with artificial data set and load testing, and don't know yet what to make of the large difference in number of defects found via the two test approaches.

We know now that making an investment in real-world simulation testing was well worth the time, and that we will continue with it as a key part of our test strategy in the future.

Testing As a Risk Management Activity

Alan S. Koch
President, ASK Process, Inc.
ask@ASKProcess.com
www.ASKprocess.com

Abstract

Managers and testers alike often describe testing as being all about finding bugs. This is a natural conclusion one might draw after observing the testing process. Look for bugs, report bugs, then ensure they get fixed. Pretty simple.

But there is a nasty problem with this. As we all know, testing all the defects out of a product is impossible in nearly all cases. And as any student of logic will affirm, even if we *could* find and dispatch every defect, it is logically impossible to prove that no more defects exist. (You can not prove the absence of something; only its presence.)

So if the objective of testing is to remove the defects from the product, we are virtually guaranteed to fail to some extent. Defects will persist.

But testing has a higher objective. Yes, testing finds defects (and of course we fix the defects we know about), but finding those defects is a means to a different – although a related end. Every product may fail in some non-trivial way that would have adverse impacts on the users or the organization. This is a risk that needs to be managed, and testing enables us to manage that risk.

Biography

Author, speaker and consultant Alan S. Koch and his company ASK Process, Inc. are celebrating 10 years of consulting with IT organizations and training IT professionals to ensure dependable, smooth, cost-effective IT Systems and Services. They capitalize on published and emerging standards to address customers' unique challenges, employing pragmatism to make customers self-sufficient.

Mr. Koch's more than three decades of experience gives him unique insight into testing from the viewpoints of tester, developer and manager. As he has come to grips with the costs, benefits, strengths and weaknesses of testing, he has progressed from merely finding bugs to viewing testing as a much more strategic activity.

The peer-reviewed journals Better Software Magazine, Crosstalk and Software Quality Professional have featured articles authored by Mr. Koch.

Mr. Koch has been invited to speak, lead workshops and provide training in such diverse places as Orlando FL, Portland OR, Zurich Switzerland, Edinburgh Scotland and Tianjin China.

IEEE (Institute for Electronic and Electrical Engineers) awarded to Mr. Koch the designation of **Senior Member**, "In recognition of significant contributions to the industry".

1. The Purpose of Testing

Software projects follow a regular progression. Requirements, Design, Code, Test, Deploy. We don't tend to give much thought to this progression or the purpose of each step. If we *do* engage is such thought, it becomes clear that most of these steps have a clear reason for being that can be used to judge their effectiveness.

- The **Requirements** phase is all about capturing a clear understanding of what should be built. It is effective if the end product is fit for purpose and fit for use.
- The **Design** phase is all about deciding *how* the product should be built. It is effective if development goes smoothly without technical surprises and issues.
- **Coding** is all about transforming the Design into executable code. It is effective if the code functions as specified in the Requirements and Design.
- **Deployment** is all about delivering the product to the customer. It is effective if the transition goes quickly and smoothly, as planned.

But what of Testing? If the purpose of testing is to find bugs, then how do we judge its effectiveness? Number of bugs found? How many is enough? How many is too few?

What if the Requirements, Design and Code steps were *very* effective? Would the lack of defects to be found mean that Testing is *ineffective*? What if the Requirements, Design and Code steps were *ineffective*? Would the multitude of defects to be found mean that Testing is *more* effective?

Ultimately, if the purpose of testing is to find defects, then every defect that escapes Test is a failure of the testing activity. If you do not find them all, then you have failed. And the more defects that testing misses, the "worse" the testing fails. But how do we know how many defects we missed? We cannot know at the time. We come to understand this very important metric only after the product has been in use for some period of time.

The reality is that testing is a filter; it will detect only a certain percentage of the existing defects, and the rest will escape. The major determiner of defect density of the product *after* test is defect density *before* test. More defects to be found means that we not only *find* more defects in Test, but we *miss* more as well.

All of this can be very depressing – *if* testing is all about finding defects. But the purpose of testing can and should extend far beyond finding defects. To discover how, we need look no further than the needs of Management.

2. What Management Really Needs to Know

As testers, we report lots of metrics to management. Numbers of defects. Defect arrival rate. Defect backlog (number of defects found but not yet fixed). Number of tests run. Number of tests passed. Percent of tests failed. And on and on.

Sometimes they ask us for this data, but more often we report these things because it is the best way we can think of to report status to them. But what do they *really* want to know? What is the burning question behind their desire to see these data (or whatever data they ask for)?

"When will it be ready to deploy?"

Of course, the key word in this question is "ready"! What does it mean for the product to be "ready" to deploy? What constitutes "ready"? If our job is to find the defects, then one might interpret "ready" to mean defect-free. (This interpretation is a common problem when managers do not understand the nature of defects in software.) But we know that defect-free is not a realistic expectation, and is most certainly beyond our reach, given our limited budgets and time.

So, if “ready” does not (or should not) mean defect-free, then what is management asking us when they pose the question, “When will it be ready?” In order to understand, we must look at it from their perspective.

The calculus of management can be boiled down to four R’s:

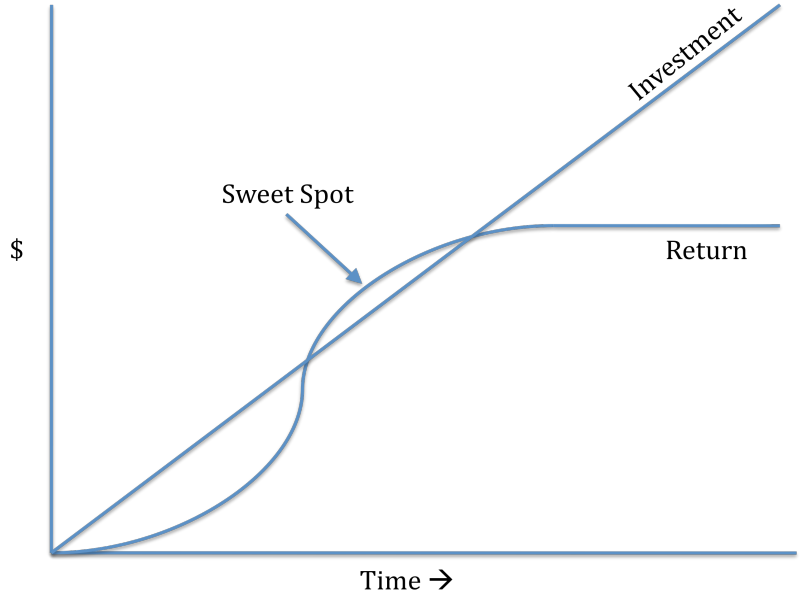
- Requirements: What we need to achieve
- Resources: The investment we must make to achieve the Requirements
- Returns: The benefits or payback on those invested Resources
- Risk: The things that get in the way of those Returns

At the point in the project when they are asking when the product will be ready to deploy, they have (hopefully) understood the Requirements, they have invested the Resources, and they are anticipating the Return. What they are asking about is the Risk.

“If we deploy today, what is the Risk that problems will reduce the Returns we expect to realize?”

“If we delay deployment and invest more Resources (time, effort, money), will it reduce the Risk and give us a better certainty of the Returns?”

Management is always looking for the “Sweet Spot” where the Return exceeds the Resources invested. The big wildcard in the whole calculus is Risk. Without Risk, Return on Investment (ROI) is easy to compute. Management could run the numbers, create a graph like the one below, and determine precisely when to stop making investments.



But risk changes the shapes of those two lines in unpredictable ways. So making the right choice about when to stop making investments requires a good understanding of the Risks. That is what they are looking to us as testers to help them to understand.

If we can help them to understand how much risk there is, they can do a better job of predicting if they have reached that Sweet Spot. If we can be more precise about risk (e.g. what types of risk remain, or what parts of the product constitute risk), then they can make even better estimates!

The good news is that we are in a very good position to provide the information that management needs to make good choices. And to the extent that we understand our role as advisors on the subject of risk,

we will be able to plan and tune the testing process so we can provide the critical information that management needs.

3. What testing can tell us about risk

Let's take a look at the types of things that our job as testers can help us to understand about risk in a software product.

When we plan our tests, we make choices about they types of tests to run and the conditions under which to run them. We know that we will not be able to test every possible set of inputs or conditions, so we try to choose those conditions and inputs that will give us the most information. Because a test that fails tells us more about the system under test than a test that passes, we have a natural bias toward tests that we judge to be more likely to fail. We are probing for defects, looking in the most likely places.

Our job is not unlike that of an exterminator brought in to determine if there are bugs in a house. The exterminator will spend little time probing hard materials like concrete and brick because few bugs are likely to be found in those conditions. Instead, he will focus on soil and wood, and especially areas of moisture. Those are where the bugs will be (if indeed there are any in the house). Testers, like exterminators know where to look for bugs, and the more experienced testers are, the better they become at knowing were to look.

Research has shows that bugs in software are indeed much like those in houses in that they tend to cluster together. Any exterminator will tell you that if you see one bug, there is a high likelihood that there are hundreds more nearby. The same is true of software, not because software bugs breed, but because they result from the same root causes. Root causes like:

- Complexity of functionality
- Errors in design
- Oversights by the developer
- Lack of knowledge or skill in development

Armed with this understanding, we can use the results of our tests to make hypotheses about the existence of bugs related to the tests we just ran. Specifically, after running a suite of related test cases, we can posit these sorts of hypotheses:

- If the tests resulted in no defects found, then we can posit that the likelihood of defects that we failed to detect in that particular part of the product is relatively low.
- If the tests revealed one or two defects, then we might suspect more in that particular part of the product and look for ways to confirm that suspicion.
- If the tests revealed many defects, then we can posit that the likelihood of even more defects in that particular part of the product is relatively high.

So, what does this tell us about risk? One of the two dimensions of risk is the likelihood that something bad will happen. Since undetected defects in the product are clearly bad, the fact that we can postulate about their likelihood means that we have an important handle on risk. This kind of handle is valuable to managers.

The other dimension of risk is the impact if that bad thing comes to be. Testers often have good insight into this as well, and that along with our understanding of likelihood constitutes a valid assessment of risk. (And even if we cannot evaluate the impact of those likely defects, management or others will be able to fill in that dimension, making our assessment of the likelihood quite valuable!)

So, we test where we suspect defects may be lurking. If we find defects there, we suspect an infestation and distrust that part of the product. Conversely, if we fail to find defects where we suspected they would be, it strengthens our confidence in that part of the product.

4. Risk-Based Testing: A Test Planning method

Knowing how to translate test results into an assessment of risk is powerful. But we can tune our test plans to make this information about risk even more valuable to management by adopting a test planning technique called “Risk-Based Testing”. This is a method that starts the test planning process with an assessment of risk and focuses testing in the areas of highest risk.

The first step in Risk-Based testing is to identify the types of risks inherent in the product. There is a wide diversity in these types of risks, but they tend to fall into two broad categories:

1) Risks that are likely because of how development is done. Some examples are:
- New technology may not function as expected
- Technology or techniques that are unfamiliar to the development team may be misused
- Less knowledgeable or experienced developers may make mistakes
- Schedule pressures may cause developers to take shortcuts
- Changing scope can invalidate design decisions causing complex workarounds in the code
- Resumption of interrupted work can result in oversights
- Changes in staffing can result in developers working with code they do not understand
- Changed code is more likely to fail than new code
- New code is more likely to fail than un-touched code

2) Risks that will have a serious impact on the customer or user of the system. Some examples are:
- Portions of the product upon which the customer will depend for critical business functions
- Failures that could go undiagnosed in production resulting in long-term impacts
- Failures that would have catastrophic impacts (e.g. loss of life or significant financial harm)

The second step is to rank the risks against each other by estimating both the probability and the impact of each. (The most important risks are those that are both highly likely and high-impact.) This results in a prioritized risk list that becomes our guide for our test plans.

The third step is to plan testing based on the prioritized Risk list. The highest-priority risks should be tested as early as is practical in the testing phase, and the lowest-risk items should be tested last. This has three beneficial effects:
- First, it ensures that the higher-risk parts of the product receive un-rushed attention during a time in the testing phase that is least chaotic.
- Second, it ensures that if there are problems in that area of the product, the development team will have plenty of time to address and correct them before deployment.
- And third, it ensures that the tests that may be skipped when time crunches at the end of the project will be the ones that are associated with the lowest-risk parts of the product.

In addition to being addressed early, the high-risk areas should also receive the most attention. So we will plan to test those areas more thoroughly. We will create more test cases for them, and will be sure to test more combinations of inputs and conditions.

Finally, when we are actually writing our test cases and creating our test data, we will do so with the nature of each risk clearly in mind. We will tailor each test to ensure that the risk that we have identified is actually tested so we can be sure that the test results provide valuable information about the risks we have identified.

Also, as each test suite and test case is written, it should be tagged with the identifier and priority of the risk that it is designed to address. Having this information at our fingertips will help us to quickly evaluate the implications of the test results, whether the test passes, fails, or is not run at all.

Risk-based Testing is powerful not only because of how it focuses our testing effort, but also because it provides key information we need to interpret our test results and provide management with information about risk.

5. When you find more defects than expected

Of course, we always expect to find defects during testing. But in some projects, there comes a time when we have found *too many* defects – when we begin to believe that the product is buggy. As we observed earlier, finding many defects in a particular part of the product indicates that more are likely to be lurking.

Clearly, this is an area of concern because the risks associated with deployment are greater. With more defects lurking in the product, there is a greater likelihood that users will be adversely impacted by them. This understanding of risk alone is invaluable. It can be the information that management needs to decide to make some additional investment to try to mitigate the risk.

But when we have used a Risk-Based approach to our test planning, we have even more information available. Will the part of the product that is suffering from failures have a high impact on users? If so, the increased likelihood of defects combined with the high impact of the functionality means that this is a high-risk situation. If management is aware of the severity of the risk, they may make different choices about making additional investments of resources to mitigate that risk.

By the same token, if the risk of failure is in a lower-impact part of the product, management will probably be more comfortable with living with the risk, even though the likelihood of problems is high. Again, the additional information is a goldmine for management decision-making.

Of course, when a product has many defects, testing tends to take longer so that there will be some significant number of planned tests that have yet to be run when the scheduled end of testing arrives. This is very much like having your testing schedule compressed, which we will address next.

6. When test time is cut short

A common situation we testers find ourselves in is compression of testing time. Development went over schedule and the product was delivered to test late, but the final delivery date remains unchanged. We are told to test “more quickly” to meet the delivery date. Sometimes this mandate is arbitrary, but other times there are good reasons why the delivery date must be honored.

Taking a Risk-Based approach gives us the most appropriate way to respond to such an edict. We can articulate to management the risks that are involved in the compressed testing schedule as compared to the originally planed one. We do this by assuming that we will execute our risk-prioritized test plan up to the point when time runs out.

If we have been instructed to work overtime, then we can adjust our plan to account for the additional effort. Just beware of the diminishing productivity of overtime. If we are to work 50% more hours per week than normal (e.g. 60 hours per week instead of 40), we should assume we will get 40-45% more work done in each of the first few weeks. If this pattern is to continue for a month or more, the additional productivity will begin to drop until after 2-3 months, productivity of the 60-hour weeks will approximate our normal 40-hour productivity.

After adjusting our plan for overtime (if needed), we can forecast the tests that we will not have time to run. Each set of functionality that we had planned to test but will not have time for represents a risk of defects. And since we already assessed the risk involved with each, we are already armed with the information that management needs. We must give them this risk information.

7. Go? Or No-Go?

Although management may be asking us when the product will be ready to deploy, in most organizations that decision is not ours to make. They aren't actually asking us to make the decision; they are asking us for pertinent information that they will use to make it.

It is our duty to bring this information about the risk of deployment to management so they can make an informed decision about the project schedule. If there is flexibility in the end-date, they might choose to allow for additional testing to reduce the risk of deploying a defect-ridden product. Then again, they may choose to accept the risk and stick with the original schedule. That is their decision to make. Your job is to give them the risk data they need to make an informed decision.

(If you are in an organization that gives you as the tester the authority to stop product releases, then you should use this information precisely as a manager would. You should assess and understand the risks and make appropriate decisions based on that information.)

If there is *no* flexibility to extend the schedule, this information is still important. When management knows of risks, they will often have a variety of ways to mitigate their effect. Even if the schedule is fixed, there may be other ways in which that information can be valuable to them. So again, it is our duty to bring it.

When we testers understand our role in the management of project risks, we can plan our testing process and report status in a way that puts important risk information into the hands of the decision-makers. And this will result in better decisions and ultimately better project results.

Test Environment Configuration in a Complex World

Authors
Liu Hong liuhong@microsoft.com
Edward French edfrench@microsoft.com
Maxim Markin maxim@microsoft.com

Abstract

Imagine you need to reproduce a complex test environment configuration that consists of Windows Server and Client machines and requires several Active Directory Domains, DNS and/or DHCP roles installed on some of the server machines. How will you implement this in a way that is robust across all available versions, editions and languages of Windows Server? With the invention of PowerShell, Windows got command line usability and tools for local and remote management of Windows Server that allows tackling these kinds of problems.
This paper offers an approach that allows capturing and reproducing a complex test configuration in an Xml file and executing a test scenario configuration using PowerShell scripting and PowerShell remoting. This approach may be used to test applications, networking components, or products which interact or run under Microsoft Windows.
In this paper we will examine the problems associated with the current approach, show the new approach, then show how the new approach is more flexible, faster to develop new test scenarios, and more robust.

Biography

Liu Hong is a Lead SDET in Microsoft Platform and Tools division at Microsoft Cooperation. She has worked at Microsoft since May 2006. She holds a Ph.D. degree in Civil Engineering from Clemson University. Before her career at Microsoft, she first worked at Aon Cooperation as a research engineer and afterwards as software developers in various companies, such as Infomove and Applied Inference in Puget Sound areas, and immediately before she joined Microsoft, as an independent contractor for Gray Hills Solutions.

Edward French have worked at Microsoft Corporation for 10 plus years as a Software Developer in Test. Hi is in the Windows Server division testing management products. He has one US patent for software development.

Maxim Markin is a Software Design Engineer in Test in Microsoft. He enjoys working together with Liu and Ed.

1. Introduction

In today's world, system administrators are required to manage more and more servers/machines that are located in a more and more complex environment, various domains and workgroups, locally and remotely. Server management products are therefore required to provide the ability for administrators to deal with the challenges they are facing. While products are made to enable the capability of managing many remote servers and machines, it also brings a new set of challenges for us as software quality assurance engineers to test these products.

One of the most important aspects of the challenge is to capture and reproduce the test environment setup and configuration the test environment in order to execute our tests. The test environment we are configuring consists of multiple computers running Microsoft Windows and the network connecting them. Configuring the networking components like switches and routers is not in the scope of this approach. The actual physical machines may be located locally or remotely and may be physical or virtual machines. They may reside in different domains and workgroups. To properly set up such an environment is a challenging task.

Microsoft Baseline Configuration Analyzer (MBCA) is a good example of multi-machine management application. MBCA is a product which helps maintain optimal system configuration by analyzing the configuration of a computer or set of computers, that are located in a distributed multi-machine environment, against a predefined set of best practices, and then reporting results of the analysis. For MBCA testing, there are five major scenarios that we focused on. They are:

1. MBCA host machine and a remote target machine
2. MBCA host machine and a number of remote target machines
3. MBCA host machine, a remote target machine, and a 2nd hop machine
4. MBCA host machine, a remote target machine, and a remote target machine
5. MBCA host machine, a remote host and remote targets, and 2nd hop machine

For MBCA scenarios, we have basically four major roles, MBCA host machine, remote host, remote target and 2nd hop machine. MBCA host machine hosts the MBCA application. Remote host is needed in case of using MBCA remoting feature to remote to the remote target. Remote target is the machine in which the configuration is to be scanned. The 2nd hop machines are the machines that remote target need to gather part of configuration data from.

Take scenario 5 as an example. Two domains and a workgroup are involved in this topology. Each domain has a single domain controller (DC) and two joined workstations which may have some Server Manager roles installed. Other two machines are joined to a workgroup and also may have some Server Manager server roles installed. The follow diagram captures scenario 5. The MBCA host machine, remote host, remote targets, and the 2nd hop machine may be any of the machines in the diagram.

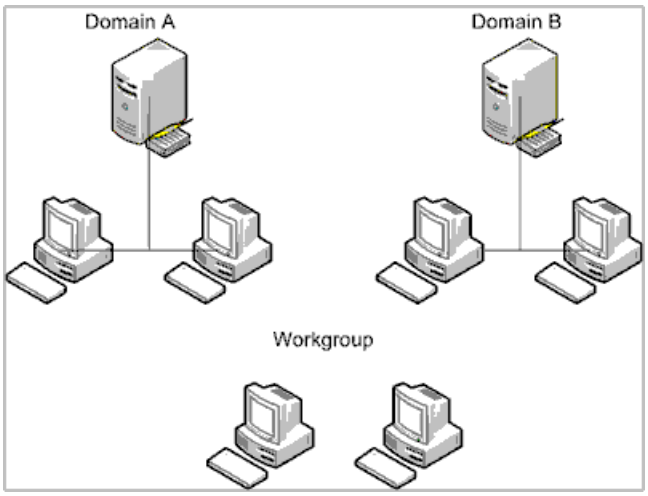


Figure 1 Sample scenario topology

In order to set up this environment, we need to accomplish the following major tasks:

1. Create two domains with one domain controller each.
2. Add two machines to each domain.
3. Create a trust-relationship between the two domains.
4. Add two machines to a workgroup.
5. Enable remoting on all machines.
6. Install MBCA and PowerShell.
7. Install server roles/components on remote machines.

Now we will discuss what we need to do to configure the test environment.

2. What We Used to Do

In single machine testing environments, we could configure a static topology, and then join the machine to be setup for testing to various places in the topology. Even if the topology is complex, the application under testing was only concerned with the single machine. Multiple individual machines could be set up quickly in this static environment.

For the challenges we are facing, we are managing multiple machines configuration where the application under test interacts with applications on other machines in the topology. This means a static set of machines can't be used for testing because each machine in the topology may be changed by the application under test. Traditionally, test machines were configured using a combination of scripting languages, such as VBScript and command line commands. For each step, we created a corresponding script or command. We then wrapped each script or command as a "job". We used these jobs in our test execution environment. For simple test execution, such as single server management, this approach worked fairly well as it was simple and straight forward.

But for the complex test setup which involves multi-machines, with the increase in the number of machines and the complexity in configuration, using this approach for configuration became cumbersome and very time consuming. In our recent MBCA testing, to cover the scenarios we talked about previously, we ended up with close to 100 "jobs". Before test automation execution, all jobs were tested separately as a first step. For each topology, these jobs were then combined together as a workflow for execution of test automation. The integration testing of this workflow creates another challenge. If the workflow failed, lots of trial and error was needed to determine the failure point because of the lack of debug ability of the scripts and commands. Configuring and debugging consumed lots of time and efforts. Needless to say we required many workflows for different scenarios. All these workflows had to be tested thoroughly for our test execution. The cost just increased linearly.

As a result, we found this approach does not scale for testing in complex test environment. Today's test environment setup is no longer single machine, but multi-machines with cross machine configuration requirements, such as MBCA testing. We need our system to scale for the new challenges we are having. Our mission is to find an approach that can reduce the cost of creating, managing and maintaining test execution, and also improve the debugging experience of configuration tasks.

3. New Approach – Integrated solution of XML and PowerShell

The new approach is to capture the test environment in a set of library files, then recreate the environment using Windows PowerShell. This new approach is based on two key technologies, XML and Windows PowerShell. XML is used to describe the configuration/execution requirements, and orchestrate automation execution. PowerShell scripts are used to parse the XML, configure, and set up the test environment.

Windows PowerShell is a command-line shell and scripting language that is designed for system administration and automation. Built on the Microsoft .NET Framework, Windows PowerShell enables IT professionals and developers to control and automate the administration of Windows and applications. Some of its 2.0 features, such as Remoting, Integrated Scripting Environment, and Scripting Debugging make it the ideal technology for the implementation of our approach to configure complex melt-machine test execution environment.

This new approach enables us to configure the test environment for cross-machine testing more efficiently. It provides a general approach of implementing multi-machine configuration in test lab and provides a common vocabulary for exchanging test configurations with external teams.

Below is an example of a PowerShell script that connects to specified machines and creates an object to be consumed by other script methods. This script gathers machine configurations by querying WMI providers.

```
function Discover-Machines ($machineNames){
    # Create script block to be executed on remote machines. The
script block will return
    # an object containing OS settings
    $scriptBlock = {
        $win32Obj = Get-WmiObject Win32_OperatingSystem
        $arch = $env:PROCESSOR_ARCHITECTURE
        Add-Member -InputObject $win32Obj -MemberType NoteProperty -
Name "Architecture" -Value $arch -Force

        $currentCulture =
[System.Globalization.CultureInfo]::CurrentUICulture.Name
        Add-Member -InputObject $win32Obj -MemberType NoteProperty -
Name "CurrentCulture" -Value $currentCulture -Force
        return $win32Obj
    }
    # Initialize the variable to hold the results as an array
    $results = @()
    # Iterate over the machines and populate the results variable
    foreach($machine in $machineNames)  {
        $session = New-PSSession -computername $machine -cred $PSCred
        if (!$?) {
```

```
            throw "Failed to connect to some of the machines.
"+$error[0]
        }
        $result = icm -Session $session -ScriptBlock $scriptBlock
        $results += $result
        Remove-PSSession -Session $session

$result.BuildNumber,$result.ProductType,$result.OperatingSystemSKU,$r
esult.OSLanguage,$result.Architecture)
    }
    $results
}
```

Figure 2 Sample PowerShell script

3.1 Configuration file

As previously mentioned, we capture test configurations in a configuration file in XML format. The configuration file can contain several test configurations or test scenarios. Each test configuration completely describes test scenario by specifying required test resources and scripts/tasks to be executed on each resource. Test resource is a machine described by number parameters: Operating system platform, edition (SKU), language, platform architecture (amd64/x86/ia64), virtual/physical machine. Then the configuration file references scripts that need to be executed on each resource. Scripts are grouped in logical units which are called test roles. A single test role may be applied to a number of resources. A test role may also have a number of parameters specified in the configuration script.

Below is the sample scenario configuration file.

```
  <Scenario Name="TwoMachineTestDemo">
    <Roles>
      <Role Name="DomainController_PDC">
        <Tasks>
          <Task Name="taskPromoteDomain"/>
        </Tasks>
      </Role>
      <Role Name="Domain1Joined">
        <Tasks>
          <Task Name="taskJoinDomain">
            <Parameter
Name="domain">"D"+$script:Machines.Domain1_DomainController</Parameter>
            <Parameter Name="user">$global:LogonUser</Parameter>
            <Parameter Name="password">$global:LogonUserPassword</Parameter>
          </Task>
        </Tasks>
      </Role>
    </Roles>
    <TestSuites>
      <TestSuite Name ="Test1" Harness="TestFramework.exe">
        <Parameter Name="-AnalyzerAssembly">'Tests.dll'</Parameter>
      </TestSuite>
    </TestSuites>
    <Machines>
      <Machine Name="Domain1_DomainController" Platform="WS08-R2"
SKU="Enterprise Server Core Edition" Language="en-US" Architecture="amd64"
ComputerName="">
```

```
        <SetupRole Name="DomainController_PDC" Status=""/>
        <TestSuite Name ="Test1" Order="0"/>
      </Machine>
      <Machine Name="Domain1_MemberServer_1" Platform="W2K3" SKU="Server
Edition" Language="en-US" Architecture="x86" ComputerName="">
        <SetupRole Name="Domain1Joined" Status=""/>
      </Machine>
    </Machines>
  </Scenario>
```

Figure 3 Sample scenario configuration file

The file captures two machine test scenario: one is Domain Controller while another is joined to that domain. Also it defines roles properties of machines that are required of each test roles. And finally it defines which tests need to be run on each resource.

3.2 Single entry point to execute scenarios.

We have a single script to drive overall execution. This script calls helper functions from other scripts. We group helper functions by library scripts, for example we have libMachineSettings.ps1 and libUtilities.ps1 scripts. Library scripts are referenced by the main script or dot-loaded in terms of PowerShell. The main script has six parameters: type of action to perform, list of machines, scenario configuration file name, scenario name, and two parameters used by an internal reporting tool. Below is the execution workflow:

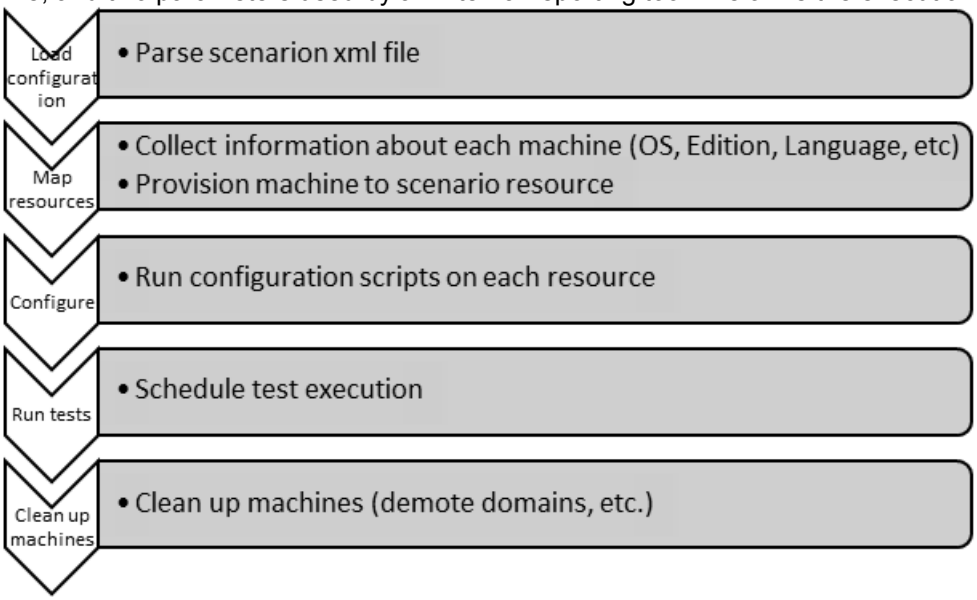


Figure 4 Scenario configuration workflow

3.3 PowerShell remoting is used to configure remote machines

PowerShell Remoting is used to control script execution on remote machines. We store credentials required to connect to remote machines in central repository.

3.4 Solution Structure

Below is the architecture layout of the solution. The solution is built on top of Windows Management Framework that includes PowerShell, Bits and WinRM. The script library of common management tasks contain library of common functions to configure a machine which includes function to join to a domain,

turn on the firewall, change IP address and etc. The top level has test roles scripts which can be applied to a test machine. By combining different test roles we get the test scenario to be executed.

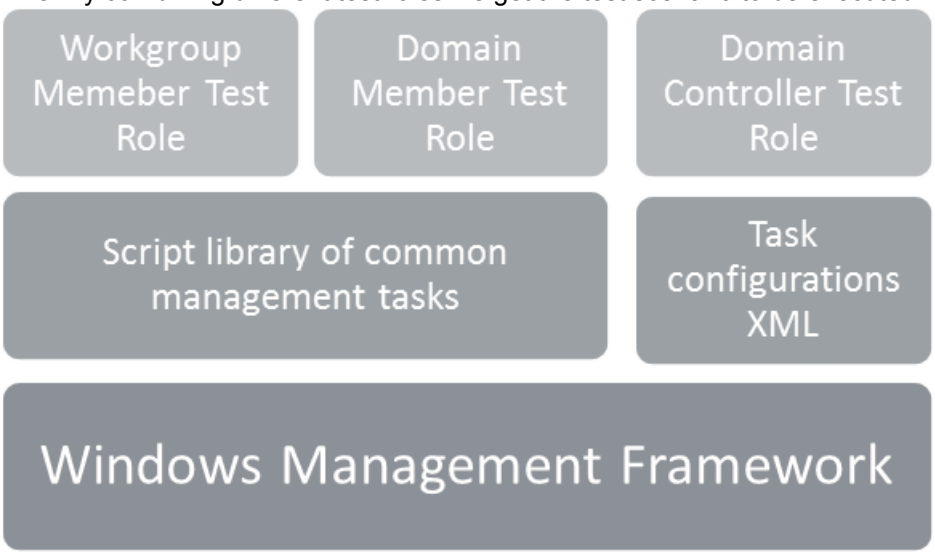


Figure 5 Architecture layout

4. Our Findings

Based on MBCA multi-machine testing, we did a comparison of test cost using the existing approach versus the new approach. As we discussed in the previous section, we have 5 different topologies for MBCA multi-machine test execution. With the assumptions that we have the basic infrastructure in place and relative familiarity with PowerShell programming, our initial finding suggested about 65% percent reduction in costing to set up and execution. The following table lists the detailed data for two approaches:

Work Items	Description	Old Approach (days)	New Approach (days)
VBScript and Command line batch files	Create, test, and debug individual task scripts	26 days (70 jobs)	N/A
PowerShell Script	Create, test, and debug individual task PowerShell scripts	N/A	10.5 (7 scripts)
Master script execution	Create, test, debug the master scripts for each scenario.	20 days (5 scenarios)	0
Write configuration file	Create xml configuration file	N/A	2
Configuration challenges	Test and debug each configuration	10	10

Process overhead	The triaging, bug fixes, communication overhead involved to get a failure in test configuration issue fixed, new test cases added to execution, fixing the bug itself.	20	5
Total		76	27.5

Figure 6 Comparison of time spend on testing

For our future Microsoft Windows testing, with the increased number of machines included in testing and configuration, we will have more topologies and more complex environment. We expect saving will be even greater.

4.1 Summary of Benefits for the New Approach

In contrast to the existing way of test environment configuration and execution, this new approach we proposed here adopted two key technologies, which when combined, bring together many benefits for efficient multi-machine test environment configuration and execution.

First, the requirements are captured in structured storage in XML document. This provides a structured and consistent way of describing configuration and execution requirements. With the help of the schema, we get the validity and integrity of the data stored in the configuration file verified. This reduces lost time when invalid entries, such as spelling errors, are made.

Secondly, PowerShell is used in parsing and programming the requirements. The advantages of using PowerShell in the new approach are:

1. Ease of coding.
2. Remote access is built-in into PowerShell: PS remoting allows remote computers to be easily managed through a SOAP-based web service
3. User experience: the user experience is the same whether you are connected to local or remote host
4. Parallel execution in PowerShell: execution of a script on remote machine does not necessarily block local execution
5. Open protocols: WinRM, the underlying protocol used by PowerShell remoting, is based on WS-Management open standard

In summary, our new test machine configuration approach provides us with the following benefits:

1. Rich debugging support of PowerShell
2. Script storage in source control system
3. Simplified maintenance of configuration scripts
4. Simplify workflow generation: New scenario can be created by generating new XML section, which describes the scenario and its configuration requirements, and combining existing scripts (tasks).
5. The scripts easily can be shared between other test teams.

4.2 Adapting to other test environments

The key to this approach is storing the configuration information in xml files. These xml files may be accessed by other technologies, but we feel PowerShell gives us the easiest tools to work with both the xml files and executing the specialized tasks required. While we're focused on creating environments that revolve around domain controllers, workstations, and network connectivity, the techniques provided by this approach are just as applicable to configuring a single machine environment to test either hardware or software.

The first step in using this approach is to model the required environment in the xml configuration file. We do this by defining a role with properties that define the tasks required to create and configure that role. We next create PowerShell scripts to execute the tasks. These individual task scripts are then executed in a master script which controls the order and on which machines the tasks are executed.

One simple pilot program would be to install an application on several machines, logon to each machine with different user credentials, then execute a set of tests on each machine. In this case, the tasks to be done might include running setup.exe for application, setting some registry values for the application, and then setting auto-logon with the user's credentials. These tasks would be added to each role definition in the configuration xml and a role would be assigned to each machine.

5. Conclusion

As we are dealing with more and more complex multi-machine test environments, the traditional approach of using scripts and commands to configure this environment will not scale. In our new approach, we capture the configuration and execution requirements in an easy to modify configuration file. With the help of PowerShell, this approach becomes a very effective way of configuring complex system and executing tests. PowerShell provides ease of implementation, ease of maintaining and ease of debugging. It provides a good solution to many of the challenges we are facing in order to configure and execute testing for multi-machine scenario effectively.

Glossary

Name	Description	Acronym
Microsoft Baseline Configuration Analyzer	Microsoft application which helps maintain optimal system configuration by analyzing the configuration of a computer or set of computers	MBCA
Active Directory	Active Directory directory service provides the means to manage the identities and relationships that make up network environments	AD
Active Domain Controller	A Windows server that maintains user database and manages security in Active Directory	DC
DNS role	Microsoft Windows role	DNS
DHCP role	Microsoft windows role	DHCP
Server Manager	Windows component for managing servers	
Windows Management Infrastructure	Infrastructure for management data and operations on Windows-based operating systems	WMI

References

Payette, B 2007. Windows PowerShell in Action

Microsoft Technet. Getting Started With Windows PowerShell
http://technet.microsoft.com/en-us/library/ee177003.aspx

Microsoft KB. Windows Management Framework (Windows PowerShell 2.0, WinRM 2.0, and BITS 4.0)
http://support.microsoft.com/kb/968929

Microsoft downloads. Microsoft Baseline Configuration Analyzer (MBCA):
http://www.microsoft.com/downloads/details.aspx?familyid=DB70824D-ABAE-4A92-9AA2-1F43C0FA49B3&displaylang=en

Issues in Verifying Reliability and Integrity of Data Intensive Web Services

Anand Chakravarty
The Microsoft Corporation
Redmond, WA 98052

Abstract

Validating the behavior of web-services in general involves the verification of the constituent parts in isolation and in integration. There are many variables to consider within each component, and these have to be verified as the system consumes growing resources under actual usage once they are shipped. These challenges multiply when such systems also have to deal with large volumes of data. In addition to the impact large data make on system performance, there are also implications on code-coverage metrics, identifying bottlenecks, predicting failure points, system versioning, data maintenance, etc. There are further challenges involved in creating a reliable set of quality metrics and measuring them within the constraints of shipping under the agile development model. A web-service that depends on huge and growing volumes of data is considered here, with an emphasis on: measuring the overall system performance under expected traffic patterns, impact of abnormal data input and system behavior under different failure conditions. The lessons learnt here have helped ship a web-service that currently serves millions of users per day, with improved data quality and following the agile development and shipping model.

Author Biography

Anand Chakravarty has been a Software Engineer in Testing at Microsoft for close to 10 years, most of it working on web-sites and web-services based products. Currently, he is on an incubation team under Microsoft Research that ships products based on Machine Translation technologies. On that team, he is primarily responsible for the stability of web-services supporting Microsoft's diverse applications that translate user input between different human languages.. This small team has been successful in shipping features and services that have seen a manifold increase in user traffic and significant improvements in linguistic quality over the past few years.

1. Introduction

Verifying the quality of the more commonly found web-services involves verifying the components of the web-service(s) individually, and then verifying the interaction of these components end-to-end. Verification of the individual components may take the form of traditional unit-tests and functional level tests. These types of tests verify the components in a more static form. While these are useful, especially if run during the development phase [1], the utility of many bugs found with such tests is significantly surpassed by verifications performed when these

components are actively running, and even more by tests run when these components are interacting with their downstream dependencies and their upstream consumers.

To such systems, the addition of components that are data-intensive introduces a multitude of diverse areas which require more rigorous verifications. These range, chronologically, from building and deploying the data to upgrading and re-versioning them. At runtime, there are the obvious performance implications of processing huge volumes of data. In addition, it is critical to ensure that the services remain reliable in terms of the data they process, resilient to data related errors and available with a high level of certainty. In addition to the testing challenge of generating adequate test data to reliably verify the system [2], the verifications performed for such a system should measure the impact of data being unavailable for any period of time, the effects of redundancy mechanisms that help keep the data synchronized across failover instances, up-to-date and online, and help in designing maintenance jobs, their impact on the execution of primary functions of the web-service and their scheduling and update notifications.

Existing research on the topic of testing web-services has involved extending the approach of testing more traditional systems to the sphere of web-services [3], applying test-driven development to web-services [1], generating pair-wise test cases for web-services [4], etc. Bochicchio *et al* [5] have presented a modeling exercise for information intensive systems.

This paper considers a case-study of a distributed set of web-services that consume huge volumes of data at runtime. These web-services together host a machine-learning system which drives Machine Translation (MT) products shipped by Microsoft Research's incubation team. These products help users translate text between different human languages, and support multiple user scenarios. The web-services use a machine-learning system that has arisen out of decades-long research in the field of Machine Translation. The entire system processes huge volumes of data, both for training purposes and during actual execution and processing of translation requests. In this paper, I present the results of verifications of such web-services in terms of their behavior at runtime when receiving requests that reasonably simulate actual traffic. While this simulation is more reliably accomplished for systems that are already live [6], there are challenges when such data are not available. In addition to estimating numbers of expected users, the test design should also account for the varying nature of user input. It is important that the load-tests are a good simulation of real user traffic, to help accurately identify memory usage patterns and stress the components realistically. While there are models that exist to create such simulations, such as the form-oriented and stochastic-form models [7], we have found that a distributed set of test requests, combined with individual test case input that is significantly larger than the average real input, leads to meaningful test scenarios that help identify key code-defects. The advantage of such an approach is that it reliably creates sufficient stress on the system to simulate conditions that are both higher than actual expected live traffic and closer to the system's overall failure point. Test results under such conditions thus help to increase confidence in the reliability of the system when it is shipped. In analyzing test results with this data, special emphasis is given to measuring system reliability, impact of performance enhancement algorithms, resilience of the system to abnormal input and recovery from failure of individual components for varying periods of time.

2. High-level architecture

The system under consideration is a distributed system of web-services consuming large volumes of data to translate text between human languages. The architecture of the system was designed to meet well-defined goals of performance, reliability and linguistic quality. Components that are data-intensive should be hosted separately and there should be separate components depending on the resources that are being consumed. Below is a high-level overview of the final design:

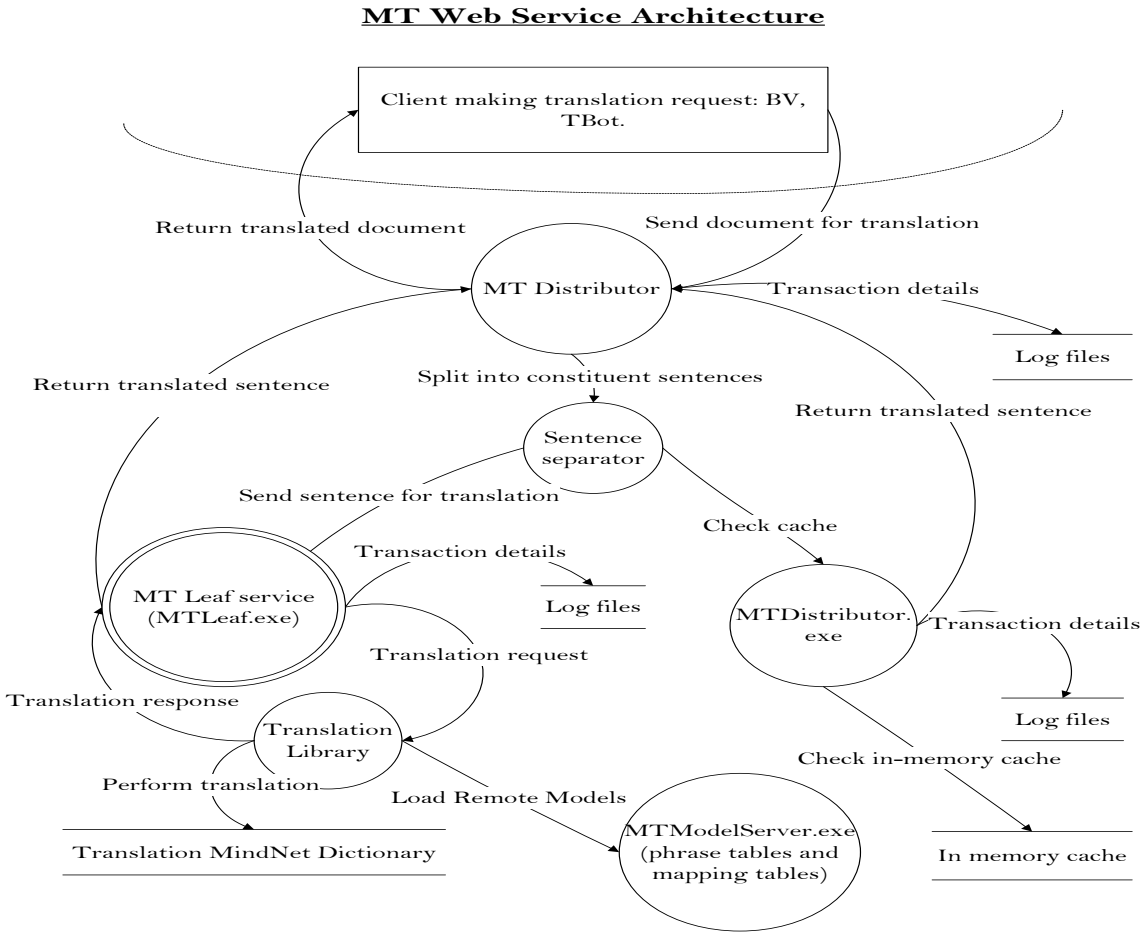


Figure 1. Web service architecture.

The data intensive components in this system are the set of web-services hosted by the leaf and the model-server machine types. These machines host translation models for each language, and are of the order of multiple gigabytes in size. In the next sections, we consider different test scenarios for these components, starting with the effects of loading multiple instances of a data component.

3. Measuring impact of different memory optimization algorithms

For applications that use large amounts of data at runtime, it is sometimes useful to load frequently used data into a memory-mapped file. These get the advantages of quick access combined with the utilization of operating system memory handling techniques. Also, when such systems are run on multi-processor systems, it is beneficial to load up multiple instances of the same data into memory and attempt to maximize the potential speed enhancements that accrue. There is however a threshold beyond which multiple instances yield little value and in fact begin to adversely impact system performance. To identify this for the Microsoft Translator-MT system, we ran the leaf processes with different settings. In one configuration, the leaf processes would load 2 instances of a required file into memory, then 3 and 4. Load-tests were run for each of these configurations. For the configuration where 2 instances were loaded into memory, the average throughput was about 5,000 words per second. When the same load-test was then run for a configuration where 4 instances of the model files were mapped into memory, the throughput went up, as expected, to 8,000 words per second.

However, the increase in throughput was offset by an increase in the number of errors that the system returned at runtime. This is a consequence of increased memory usage when more instances are mapped to memory, which results in higher page faults and consequently more errors. Below is a comparison of test results for one of the language pairs that was stressed:

Table 1. Comparison of failure rates

	With 2 instances of memory mapped file	With 4 instances of memory mapped file
Throughput, in words per second	5,000	8,000
Percentage of requests failing	1.6%	13.5%

Running load-tests with different configurations, and measuring the performance and success rate of the test cases, showed that depending on available resources and the size of data files, memory mapping soon begins to drastically increase the rate of errors beyond a certain point. In this specific case, which dealt with a service using managed .NET components, part of the overhead was in garbage collection.

The load-tests found that when 4 instances of each translation model is loaded into memory, the time spent in garbage collection by the leaf processes was between 10 to 40%, depending on the size of models. Leaf process that had larger models to load spent close to 40% of time in garbage collection, while leaf processes that had smaller translation models spent 10% of their time in garbage collection. When applications spend more than 10% of time in garbage collection, there are fewer resources available for doing real application specific work. The next section gives more details of the impact of garbage collection on system behavior.

4. Garbage collection for services

Garbage collection is a memory management method used in certain application runtime environments such as the Java VM and the .NET runtime environment [8]. While there are many benefits of this method, it is vital to measure its impact on overall system performance so as to use resources in an optimal manner. Applications that are data intensive are especially vulnerable to memory pressures that might be worsened by garbage collection (GC) operations.

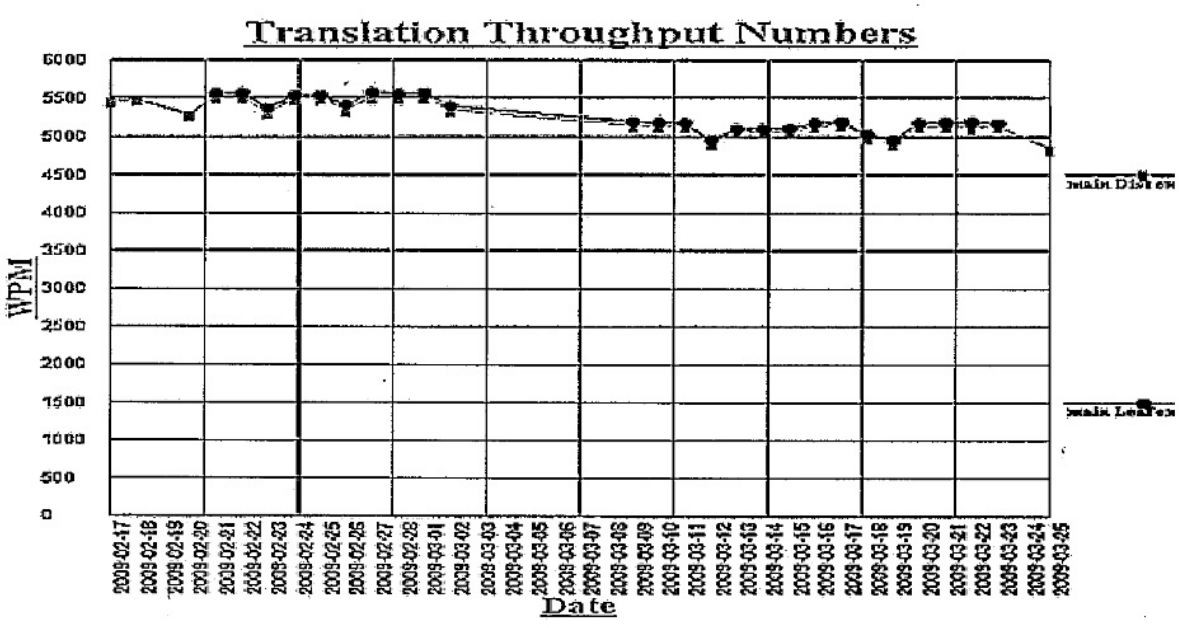
In the .NET runtime environment, there are different flavors of garbage collection, named as: workstation garbage-collection, server GC and concurrent GC. It is important to choose the right flavor to optimally use available resources while sustaining real usage scenarios. For the MT web-services, in order to figure out the right version of garbage-collection to use, load-tests were run with the constituent applications using different flavors of garbage-collection. The results for such a load-test run with the web-services using Workstation-GC showed that the web-services spent close to 20% of their time in garbage collection. The same system when run with Server-GC enabled saw less overhead, only about 4% on average.

The percentage time in garbage collection reached close to 20% during this round of tests. The same system when run with Server-GC enabled saw less overhead, only about 4% on average. There was also a difference in the pattern over time of garbage collection. The overhead was reduced when Server-GC was enabled, compared to the more commonly used Workstation-GC. Reducing this overhead is especially valuable for data-intensive systems which use large amounts of memory or disk-space. Resources thus freed up are diverted to more meaningful data-processing tasks and consequently yield benefits in overall system performance.

5. Overhead of individual components

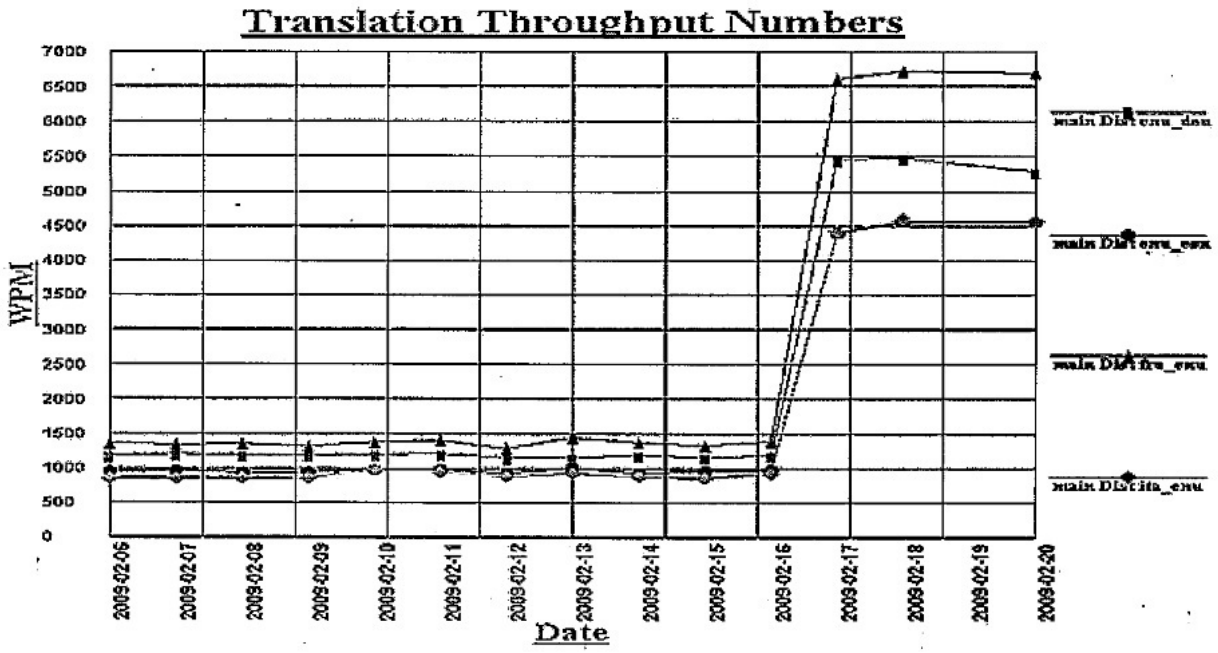
Identifying the bottlenecks in a distributed system is vital to ensuring efficient use of resources [9]. System bottlenecks are most commonly found in components performing I/O heavy tasks [10]. The architecture of the MT Translation web-service involves communications between multiple machines to completely process a translation request. This requires each component to be lean and extremely responsive to other components. Each component has its own communication ports, its own queues, different rates of execution, and different ways of failing. Our stress tests helped to measure these overheads, and as the system was made more efficient, they helped to prove that in all this interaction, the overhead in going across multiple machines was minimal.

For these tests, we start with a tightly defined set of input test cases, and hit the individual components separately. For example, we send translation requests to the distributor machines, and then the same set of translation requests directly to a leaf. The leaf is one step closer to the model-server, so it is expected that performance is faster when requests are sent to a leaf, as opposed to when they are sent to a distributor. As seen from Figure 6, the eventual implementation of these 2 components was good enough that there was almost no overhead involved in going through another machine type, other than network latencies. The darker line in Figure 6 is the throughput when test requests are going directly to a leaf machine, and the lighter line is the throughput when the same requests go through the distributor, which is an extra layer compared to the first scenario:



The tests that generated the above graph were run regularly, across rolling builds, thus enabling easy identification of changes that affect performance. Below is an instance where a particular code-change resulted in a 4-fold performance boost across multiple language pairs:

Figure 7. Quantifying impact of code-changes



Designing a test-case system that allows the automated monitoring of individual components and reports results that easily highlight the impact of system changes is essential to evaluating end-to-end performance of the distributed components of web-services.

6. Verifying Resilience to failure

It is imperative that web-services have a high rate of availability. While hardware redundancy is a good first step to achieving high availability, it is not price-optimal. Also, even with hardware redundancy, it is vital for all components of a web-service to understand the availability of the components they depend on, recover from the failure of these components by quickly identifying fallback paths upon failure, and report failures in an actionable manner when bottle-necked.

In addition to overall system resilience, there are also aspects of the behavior of individual components in their interaction with other parts of the web-service that affect resilience. For the MSR-MT system, with its distributed web-services, with different services having different resource constraints, tests were run with the individual components in different states of failure to identify such aspects of system behavior.

Assuming there are multiple machines supporting a particular component of a web-service, consumers of that component should have a clear idea of their availability. For example, in the MSR-MT system, if a model-server is assumed to be running on 4 different machines, and one of them experiences a failure, the leaf and distributor processes that depend on it should remain unaffected, apart from performance issues owing to reduced resources.

Verifying this scenario for the MSR-MT web-services resulted in the finding of bugs relating to process initialization, recovery and cleanup. It is important that these issues are validating by executing failure scenarios, to ensure the web-service remains responsive to user despite transient failures.

7. Conclusions

The challenges of testing web-services are compounded when there are data-intensive aspects of these services. In order to properly verify the impact of such data, it is important to define metrics that quantify the resource-usage of the overall system. Properly simulating actual user traffic is vital, as is verifying the system under conditions of failure. Validating design decisions by providing test results that show the performance of the system under different configurations, especially when these configurations impact resources such as processor or memory usage, helps to implement a system that optimally uses those resources. As shown in our investigations of garbage-collection, it is critical to verify the system considering the resource-management feature of the platform on which the web-services are built. The identification of system bottlenecks by focused testing of individual components results in a system that is combines the benefits of modular design of components with a high-performing, efficient overall system. Such an approach to testing has helped the MSR-Machine Translation team ship a system of web-services that have supported a rapidly growing set of users and features, with increasing performance and quality.

8. References

[1] Nuno Laranjeiro, Marco Vieira, "Extending Test-Driven Development for Robust Web Services," Dependability, International Conference on, pp. 122-127, 2009 *Second International Conference on Dependability, 2009*

[2] Ying Jiang, Ying-Na Li, Shan-Shan Hou, Lu Zhang, "Test-Data Generation for Web Services Based on Contract Mutation," ssiri, pp.281-286, 2009 *Third IEEE International Conference on Secure Software Integration and Reliability Improvement, 2009*

[3] Samer Hanna, Malcolm Munro, "An Approach for Specification-based Test Case Generation for Web Services," aiccsa, pp.16-23, 2007 *IEEE/ACS International Conference on Computer Systems and Applications*, 2007[4] Siripol Noikajana, Taratip Suwannasart, "An Improved Test Case Generation Method for Web Service Testing from WSDL-S and OCL with Pair-Wise Testing Technique," compsac, vol. 1, pp.115-123, 2009 *33rd Annual IEEE International Computer Software and Applications Conference, 2009*

[5] Mario Bochicchio, Antonella Longo, "Conceptual Modeling of Data Intensive and Information Intensive Web Applications," mmm, pp.292, *10th International Multimedia Modelling Conference, 2004*

[6] Ka-I Pun, Kin-Chan Pau, Yain-Whar Si, "Modeling Support for Simulating Traffic Intensive Web Applications," wi-iat, vol. 3, pp.512-516, 2008 IEEE/WIC/ACM *International Conference on Web Intelligence and Intelligent Agent Technology, 2008*

[7] Lutteroth, C.; Weber, G.; , "Modeling a Realistic Workload for Performance Testing," *Enterprise Distributed Object Computing Conference, 2008. EDOC '08. 12th International IEEE* , vol., no., pp.149-158, 15-19 Sept. 2008 doi: 10.1109/EDOC.2008.40

[8] Tomas Kalibera, Filip Pizlo, Antony L. Hosking, Jan Vitek, "Scheduling Hard Real-Time Garbage Collection," rtss, pp.81-92, 2009 *30th IEEE Real-Time Systems Symposium, 2009*

[9] Petriu, D.; Shousha, C.; Jalnapurkar, A.; , "Architecture-based performance analysis applied to a telecommunication system," *Software Engineering, IEEE Transactions on* , vol.26, no.11, pp. 1049- 1065, Nov 2000

[10] Kallahalla, M.; Varman, P.J.; , "PC-OPT: optimal offline prefetching and caching for parallel I/O systems," *Computers, IEEE Transactions on* , vol.51, no.11, pp. 1333- 1344, Nov 2002

Peering Into the White Box: A Testers Approach to Code Reviews

Alan Page
alanpa@microsoft.com

Abstract

Code reviews (including peer reviews, inspections and walkthroughs) are consistently recognized as an effective method of finding many types of software bugs early – yet many software teams struggle to get good value (or consistent results) from their code reviews. Furthermore, code reviews are mostly considered an activity tackled by developers, and not an activity that typically falls within the realm of the test team. Code reviews, however, are an activity that questions software code; and many testers who conduct code reviews question the software code differently than their peers in development.

This paper will present how a test team at Microsoft used code reviews as a method to improve code quality and more importantly as a learning process for the entire test team. The paper will also discuss how smart and consistent application of lightweight root cause analysis and the creation of code review checklists led the path to success – and how any team can use these principles to reach these same levels of success.

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 Communicator team. In his career at Microsoft, Alan has worked on various versions of Windows, Internet Explorer, and Windows CE, and has functioned as Microsoft's Director of Test Excellence. Alan writes about testing on his blog (http://angryweasel.com/blog), is the lead author on How We Test Software at Microsoft (Microsoft Press, 2008), and contributed a chapter to Beautiful Testing (O'Reilly, 2009).

1. Introduction

Code reviews - the practice of systematically examining software source code - has been a long used practice in software development. Code reviews are one of the first lines of defense to find errors and are a “cheap” way to find bugs because of this. A second pair of eyes on anything is often one of the most straightforward ways to find an error. In my experience, code quality often improves if the author merely knows the code will be reviewed, as the author is less likely to write “tricky” code or take shortcuts.

Code reviews have a learning aspect as well. They work well for sharing techniques or algorithms, and are a great way to integrate new people onto the team or help team members learn about other parts of the project.

Finally, code reviews help a team build trust. Teams that conduct code reviews get used to the idea of analyzing the *code*, and not the person; and they get used to their *code* being critiqued and not themselves, and this gives them trust and freedom to talk about other things to improve in the product without making it a people issue.

2. The Tester Point of View

Code reviews are most often recognized as an activity done by the development team before checking in code or before passing it off in executable form to the test team. Developer code reviews are one of the farthest upstream activities there are to find bugs, thus are one of the most cost-effective methods of finding bugs.

Testing - even in the most agile of development environments, most often occur downstream from where developer code reviews happen. One notable exception is to pair a developer and a tester in a pair-programming session. Pair-programming is, in fact, a form of code review.

So why have testers perform code reviews?

On the Office Communicator team at Microsoft, our reasoning is this: our developers (and hopefully yours too) write unit tests. Unit tests are another upstream method of finding errors early and do a lot to improve code quality. However, we still test the code - on a much larger scale than unit tests are designed to do, but *our approach to testing from the test team perspective is different from the developer approach to writing unit tests.* Many unit tests, as good as they are at finding errors, test little more than the happy path. Unit tests test the code from the developer’s point of view and try to prove that the design and behavior is correct. Testers, on the other hand, ask numerous questions about the software they’re testing. Many of these questions begin with “what if?” – as in “what if we tried a really long string in this edit field?”, or “what if I try to launch the application twelve times?”. Most of these questions are well beyond the scope of what a unit test should do, and is one of the reasons we have software testers.

We apply the same reasoning to code reviews. When developers review code, they don’t ask as many questions. They make sure variable initialization is correct, that algorithms are optimal, and that the code structure and flow makes sense. The effort of review is worth it. On our team, many of the developer code reviews of significance (more than a few dozen lines) finds at least one issue or suggestion during review.

We also hypothesize that testers will ask different questions of the code than developers. When reviewing code, instead of asking questions like “will this code work correctly?” testers are more prone to ask questions like “under what situations that I can think of will this code fail?” The tester code reviews were not designed under any means to replace the developer reviews already in place, but were designed as

just another test technique for the test team. With this in mind, we set out on our experiment of tester code reviews.

3. Types of Code Reviews

Code reviews vary in style, formality, and effectiveness. The least formal type of review is where the author of the code literally grabs someone out of the hall and says, “Hey – would you take a look at this?” A step up from there (in both formality and effectiveness) is to send an email or instant message to teammates asking for a quick review. At the other end of the spectrum are formal code inspections where pre-work for each reviewer is identified, and a moderator ensures that the code is reviewed, and statistics are tracked carefully in a group meeting.

Microsoft development teams use varying degrees of formality in code reviews – typically falling somewhere between the extremes outlined above. Code reviews are a good practice for development teams and their value to product quality is significant.

Code inspections as invented by Michael Fagan[i] are the most formal method of code review, and are highly effective at finding errors in code early in the product development cycle. However, they require that developers spend a large amount of their time preparing for, and attending the inspection meetings (also in training to know how to do a Fagan inspection in the first place). In many organizations practicing Fagan inspections, the number of total person-hours required for reviews will outweigh the coding time. This is, in my experience, the primary reason more teams do not attempt Fagan inspections.

4. Our Approach to Tester Code Reviews

For our tester code reviews, we decided to try a slightly less formal approach. We wanted the reviews to be as thorough as possible while keeping time investment as low as possible and interest and engagement of the team high.

4.1 The Kickoff

We began the effort by presenting a tech talk for the test team on code reviews. Since most of the team was only familiar with ad-hoc code reviews, we presented an overview of different code review approaches and techniques in order to provide some background information.

A week later, we held another kickoff meeting for all testers on the team who were interested in taking part in this code review experiment. We did not look for any specific characteristics or skills in the participants other than a willingness to try something new and commit a small amount of time every week to the effort (note that nearly all testers at Microsoft are fluent programmers). In this meeting, we went over the details of our code review process and expectations for the volunteers. The expectations defined at the beginning of the experiment included:

- **Review for 60-90 minutes each day, and record the time spent on review.** Part of the learning process for doing good code reviews is practice. By establishing a rhythm of performing reviews, we expect testers to improve their technique. However, we’ve also found that the ability to find errors consistently in code declines after 60-90 minutes, and we also wanted each participant to discover when their own reviewing abilities started to fade within a session. Finally, we asked them to track the actual time so we could calculate yield – the number of issues they found within their review period.

- **Make a note of all issues or questions found during the review.** We used this information in the review meeting, as well as for tracking yield.
- **Schedule a regular meeting for team reviews.** We asked teams to schedule a regular meeting where they would discuss and note errors or questions found during the individual reviews. We had two primary reasons for the regular meetings. The first was to determine which issues multiple members of the team are finding, as these had a higher likelihood for being real errors that are likely to be fixed. The second, and equally as valuable was the learning aspect. The differences in issues found by different team members lead to discussion and sharing of how a particular reviewer discovers an error, which in turn leads to improvement by all reviewers in subsequent reviews.
- **Use a checklist.** We gave everyone a checklist including types of errors generally found during code review. We asked them to select one item on the checklist, and review a section of code *looking only for that single item*. (A "section" could be a single function, a class, a screen, or any other sensible partition). Looking for a single item enables a reviewer to minimize distractions and find errors more consistently. Reviewers would then have the option of either reviewing an additional section of code looking for the same class of error, or could review the same section again looking for a different error. A sample checklist is included in the appendix.

At the end of the kickoff meeting, we divided the attendees into teams (mostly representing particular feature areas) and gave each team the initial assignment of selecting the code for their first review.

4.2 Selecting the Code for Review

We did not have a goal of reviewing all code. Given the size of our product and the other testing investments we have, this was not feasible or practical. Instead, we used other factors to guide us in this decision.

We used a risk-based approach to identify potential code for review. Some of the areas we looked at to identify risk included:

- **Code Churn** – We identified areas of the product with high amounts of recently modified or added code and selected source files within that area as potentially reviewable.
- **Code Complexity** – Code with a large number of paths or variables, or code with a high number of dependencies is prone to error.
- **Tester Intuition** – "Gut feel" or "tester intuition"; testers often have a hunch of where bugs may lurk, and in our experience, these hunches are accurate enough to influence the selection.

5. Tester Code Reviews in Action

During each meeting, we began by noting how much time each reviewer was able to review the code. Next, we walked sequentially through the code assignment for the week, stopping to discuss each issue noted by the reviewers. These discussions included bugs as well as areas of confusion (e.g. details of the .Net common language runtime garbage collector). The result of much of the initial effort was in learning the product, or some of the idiosyncrasies of the programming languages (our product uses both C++ and C#).

Most teams averaged about one thousand lines of reviewed code each week. The initial stages included a lot of learning time as the reviewers got used to the idea of reviewing the code and using the checklist. We also encouraged everyone to suggest new checklist items if they found additional errors, and to

suggest removing checklist items that were not (or did not seem) effective at finding errors. We were not trying to find every single possible error, and keeping the number of checklist items manageable was important.

Testers found, on average, ten unique “issues” per thousand lines of code during review. Slightly less than ten per cent of these were bugs that could affect the customer experience of the product. The remainder of the issues were code issues potentially affecting maintenance and readability of the code. The intangible value of a maintainable code base is important enough, that over eighty-five per cent of these issues are scheduled to be addressed.

6. Lessons Learned

Test team code reviews remain in progress on our team, and we continue to tweak our approach as we learn what works and what does not work. Thus far, test team members as well as product management view the approach as successful, and as something that should continue throughout this release and into the next release of our product.

Some of the biggest lessons learned include:

- **Learning** – Initially, much of the code review effort went toward learning the product. The testers were adept in the functionality of the product, but reviewing the code opened up many of the nuances of the implementation. Although we didn’t find many bugs in the initial steps, we felt that learning the how and why of the code implementation was a benefit for the team. Several of the testers on the team found additional areas or parameters to test based on the code review.
- **Balancing the code review commitment with “other” test activities** – we also discovered that it was difficult for many of the reviewers to commit regularly to the 60-90 minutes of review per day. Each of the testers involved in the review was also responsible for their day-to-day testing work, and this “new” effort sometimes fell by the wayside.
- **Involving the development team** – we started the process attempting to have as little development effort as possible, but we changed that approach slightly. Originally, the test team performed the reviews, and then one representative from test took the results to the developer and met one on one to discuss the issues and questions. We ended up modifying that and invited the developers to attend the review meetings. This enabled us to transfer knowledge between test and development faster, and increased the ability of testers to read the code.
- **Developing specialization and experts** – reviewing code is a lot different from most other test techniques, so we did not expect everyone to become an expert. Over time, testers tend to specialize, and we expected that there would be differentiation in tester’s ability to review code effectively. In the relatively short time, we’ve asked testers to review code we’ve seen three of the participants stand out as effective code reviewers. We expect that, as we continue with this effort, that a few more stars will rise.
- **The checklist approach** – the formality of the approach worked well for the team. The use of checklists and the inspection meeting encouraged learning and participation, without adding too many extra meetings. Our goal was to keep the approach as simple and lightweight as possible within a small level of formality (e.g. checklist driven inspections, regular meetings, and a few metrics).
- **Bug tracking** – we remain excited about the learning opportunities that these reviews created, but another potential drawback of our current approach is that we do not track the bugs found during code review in our product bug tracking system. We did this purposely to keep the system lightweight, but by doing that we lose a bit of insight that we can share with future testers and

developers on the team about how the code works and what types and numbers of bugs have been found in a particular component over time from reviews.

7. Recommendations

Code reviews performed by testers have been successful on our team, and we expect that the practice will work well for any team with some obvious precursors. The need for source code access is obvious, but a level of trust between the testers and developers on the team is also necessary in order to be successful. Testers need to know that their feedback will be taken seriously, and developers need to understand that the critique is of the code, and not a critique of the developer. The testers will also need enough programming language background to be effective in reviewing the code.

8. Future Plans

We expect to continue the practice of test reviews for the long term. It has enabled our test team to find bugs in a new way and has facilitated learning across the team. In addition, we'll ask our "star" reviewers to examine much of the code introduced late in the product cycle to reduce the risk of regression or new bugs from those late changes. We'll also continue to encourage and coach teams in reviewing code so that we can continue to grow new review experts and increase the quality of our product code overall.

We will also examine how we can tie the comments and feedback from the reviews to the actual code. We expect that this will make learning the code much easier for new team members, and since we're only looking at code that we deem to be risky in the first place, chances are that any additional context we can provide with the code will be beneficial for anyone involved in developing or testing that code.

9. Recommended Reading

Software Inspection – Tom Gilb & Dorothy Graham. Addison-Wesley Professional (January 10, 1994)

The Best Kept Secrets of Peer Code Review - http://smartbear.com/codecollab-code-review-book.php

Software Inspection (web article) - http://www.the-software-experts.de/e_dta-sw-test-inspection.htm

Fagan Inspections (Wikipedia entry) - http://en.wikipedia.org/wiki/Fagan_inspection

Appendix

Sample checklist items

- Are input parameters validated and checked for null?
- All exceptions must not be caught and sunk (i.e. not rethrown)
- Do the comments accurately describe the code?
- Examine loops for performance issues
- There must be no hard-coded strings and magic numbers in the code
- All user-visible strings must be localizable (i.e. retrieved from a resource file)
- Verify correctness of operators
- Is arithmetic safe for overflow and 64-bit math?
- Are the functions or source files exceptionally large or complex?
- Are COM interfaces properly ref counted?
- If there are multiple exit points (returns) from a function, can they be combined?
- Is the code overly complex? Could it be simpler?
- Do the function and variable names make sense?
- All memory allocations should be checked for failure
- Are shared resources guarded appropriately using synch objects?
- Are threads cleaned up on exit?
- Check for precedence errors – parenthesis are free
- Are untrusted inputs validated?
- Check for properly constructed switch statements
- Are comments appropriate and useful?
- Does the code have performance implications – if so, are optimizations possible?
- Are there spelling errors in names or comments?
- Have boundary conditions been checked?
- Are there unused elements in data structures?
- Are string buffers big enough for localized content?
- Does the new code duplicate code that exists elsewhere in the project?
- Are there unnecessary global variables?

[i] http://en.wikipedia.org/wiki/Fagan_inspection

Testing Concurrency Runtime via a Stochastic Stress Framework

Atilla Gunal, Rahul V. Patil
Technical Computing, Microsoft
atgunal@microsoft.com, rpatil@microsoft.com

Abstract

The non-linear interaction of many software components makes quality assurance a complex problem even for traditional serial code. Concurrency and interaction from multiple threads add an additional dimension to software complexity. This extra dimension introduces unique bug types such as deadlocks, live-locks, and race conditions.

In this paper, we will describe how a solid stress framework, complete with integrated structured randomization and methodical meddling of temporal properties makes practical software quality assurance possible. Specifically, we discuss the methods and practices applied to provide solid assurance to a critical commercial component – the native Concurrency Runtime [3] stack from Microsoft. First, by applying random distributions in individual tests and integrating such individual tests via a statistically fair scheduler, we describe how to cope with traversing the seemingly infinite interaction patterns. Second, we will expose how such testing helps identify hangs stemming from deadlocks and live-locks. Third, we will talk about injecting randomization to the temporal properties of the software system methodically, and how that can be used to assure that we find bugs with reasonable probabilistic expectation. We will conclude with a brief survey of the effectiveness of our stochastic stress framework over other tools.

Biography

Atilla Gunal is a software development engineer in test at Microsoft Technical Computing Group. He has been thinking, implementing and improving Concurrency Runtime testing strategies since 2007. Before Microsoft, he has worked in the defense industry in Turkey for 7 years in positions ranging from Software Engineer to Systems Engineer and Technical Project Management. Atilla holds a BSc degree from Istanbul Technical University Mathematics Engineering and a Master's degree in System Analysis. He has relocated to Redmond to join Microsoft as he was pursuing his Computer Engineering PhD degree in Istanbul. His interests are test automation, concurrency testing, modeling with math and applications with software.

Rahul V. Patil is a senior QA lead for Microsoft's new native Concurrency Runtime technologies. As a QA lead in charge of a brand new concurrency platform, Rahul has had to address reporting quality on new risks introduced by non-determinism inherent in concurrent software. He also holds a Master's degree in Software Engineering and has been working at Microsoft for the past 6 years, testing various SDKs and platforms.

1. Introduction

Even with the improvements in software engineering [9], the process of developing software is still complex and error-prone. A report from Standish Group [1] indicates that software quality has improved from 1994 to 2006. However, the failure rate of software projects is still a significant 46%. Holzmann [2] argues that for traditional serial software it is impossible to provide input data with all possible variations for a complete testing. The situation is even more complex when the input is coming from multiple threads where the relative ordering of events yields different run-time behavior. With multi-core CPUs becoming prominent, parallel applications will become more prevalent and new quality risks will become a significant concern.

In this paper, we describe how we used a stochastic stress framework that aims to overcome the complexities of testing concurrent software. We will start with a brief description of the Concurrency Runtime, then define a model for testing highly concurrent software and talk about its implementation characteristics. We will focus on three parts of the model that make it an effective bug detector:

i) Ability to integrate random distributions in test code that represent user scenarios.
ii) Ability to combine multiple tests that act on shared software under test to achieve complex scenarios.
iii) Ability to randomize thread executions for finer grain interleavings.

We will conclude with a survey of bugs found with the stress framework together with a comparison of different approaches for concurrency testing.

We assume that the reader has basic knowledge about parallel programming using threads and is familiar with typical concurrency bugs. Reader is encouraged to refer to [7] for a discussion on concurrency bugs.

2. Software under test: Concurrency Runtime

The Concurrency Runtime is a C++ library that helps developers apply parallel programming in their applications under the Visual Studio 2010 development environment. The overall architecture is shown in Figure 1.

The resource manager is a process wide entity that arbitrates threads as a resource depending on the load of the schedulers. The task schedulers are long lived entities that apply smart heuristics to maximize throughput of parallel tasks. The Agents Library as well as the Parallel Patterns Library (PPL) provide an Application Programming Interface (API) that allows users to express concurrency and describe tasks in a simple way (an example is the parallel_for API that can easily parallelize a serial for loop automatically).

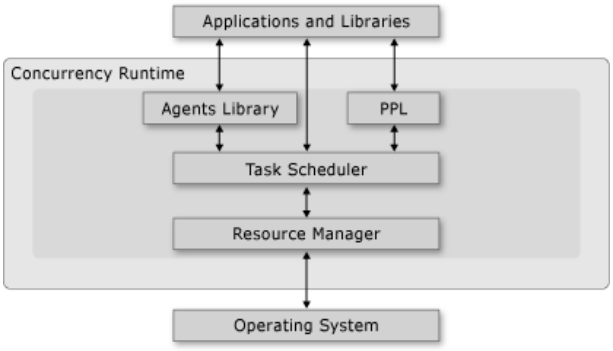


Figure 1 - Component stack of the Concurrency Runtime

The goal for Quality Assurance (QA) with respect to concurrency risks is to ensure that the runtime is free of:

- Adverse race conditions that lead to application malfunctions or crashes.
- Deadlocks that arise out of bad synchronization schemes.
- Task starvation that arises out of bad scheduling heuristics.

There are two important aspects of the runtime for our discussion around risks to concurrency testing:

i) Synchronization used within the runtime

The Concurrency Runtime has to ensure thread safety of its internal states while maintaining the goal of maximizing throughput. To achieve that, its developers applied synchronization in the following order of preference:

a) Do not use synchronization if the race is harmless (benign)
b) Use interlocked operations and memory barriers for synchronization
c) Use multiple fine grain locks
d) Use coarse grain locks

In the order of occurrence, a) has best performance with highest QA risk, while d) has the worst performance with lowest QA risk. Hence, as QA we treated the Concurrency Runtime as a huge lock free algorithm and designed tests that are able to traverse this pool of race conditions.

ii) Temporal and dynamic interaction of runtime components

The scheduler and resource management pieces of the runtime are long running, living entities of the process. They traverse many states in their lifetime, triggered by the temporal and dynamic interaction of various components and user actions. Many defects may be unique to traversing to certain states with a specific temporal precondition. Thus, it is very important to have a framework that is capable of traversing this huge state space with seemingly infinite temporal preconditions.

3. Model for Concurrency Testing

In this section, we will introduce a Concurrency Testing Model (CTM) to work with the challenges of testing concurrent software. CTM can be defined as a set of Stress Tests (ST) implemented on top of a Stress Framework (SF).

$$CTM = \{SF, \{ST_i\}\} \quad i = 1,2, \dots, TestCount$$

We can further decompose the Stress Framework (SF) into the following elements:

- Randomizer (R)
- Individual Stress Test Structure (ISTS)
- Stress Test Combiner Structure (STCS)
- Stress Test Driver (STD)
- Runtime Randomizations (RR)

$$SF = \{R, ISTS, STCS, STD, RR\}$$

A block diagram of the proposed CTM can be seen in Figure 2. The following paragraphs describe each component in more detail.

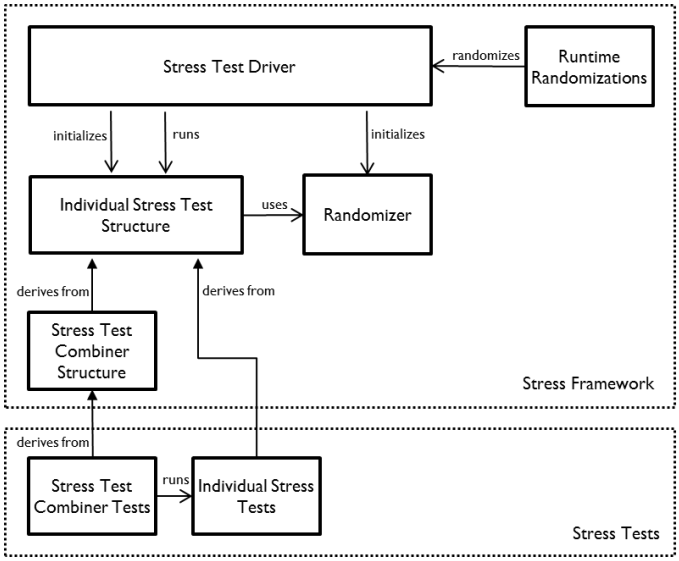


Figure 2 - Block diagram of the Concurrency Testing Model

3.1. Randomizer

Running the same static test over and over has a very small potential of exposing different concurrency bugs. Burckhardt et al. [5] for example, observed that bug finding capability of tests that depend on external factors only such as the OS scheduler or the load of the operating system instance is observed to be very low. Thus, it is essential to have an element in the model that varies the execution path. In our model that element is the randomizer which helps test developers to generate test parameters from random distributions that represent the scenario of the individual test. The parameters can be inputs to the software under test or even the control logic decisions of what action to take next.

3.2. Individual Stress Test Structure (ISTS)

ISTS defines the interface that each individual test has to implement. The interface makes it possible to implement tools such as test drivers to create, initialize, cleanup and run until cancellation.

3.3. Stress Test Driver

Given an individual test that follows the ISTS interface, the driver can execute that test until cancelled. The driver can also be used to implement common verifications and debugging support such as:

i) Hang/Starvation/Deadlock detection
The driver will monitor the individual test execution and report a hang if the test does not return in a predefined amount of time.

ii) Memory leak detection
The driver will monitor tests for memory leaks.

iii) Unhandled exception handler
The driver will capture the state of software under test upon an unhandled exception during an execution of the test. Typically exceptions result from state corruption due to underlying races.

3.4. Stress Test Combiner Structure (STCS)

This component targets concurrency bugs that can only be exercised as a result of running multiple individual stress tests simultaneously from multiple threads (i.e. by exploring the thousands of states due to the dynamic interaction of the different components, resulting from many user actions). An implementation of an STCS takes many individual tests and runs them simultaneously. STCS is only possible because there is a well-known ISTS. STCS, itself implements the ISTS. This makes the design of stress test driver simpler by being agnostic of the test type.

3.5. Stress Tests

Stress tests can be grouped into two categories:

- Tests that implement the ISTS interface (Individual Stress Tests)
- Tests that implement the STCS interface (Stress Test Combiner Tests)

3.5.1. Individual Stress Tests

Individual stress tests are scoped tests that focus on a specific use case. These can also make up the building blocks of more complex tests. Individual stress tests can further be examined in two categories:

i) Stateless

Stateless tests have a repository of objects of software under test and associated actions on objects. The actions are executed according to a random distribution. Such tests are defined as:

$$Stateless = \left\{\left\{O_i, A_{ij}\right\}\right\} i = 1,2, \dots, ObjectCount, j = 1,2, \dots, ActionCount$$

Where O_i denotes the object under test and A_{ij} denotes the actions applicable on i^{th} object.

Stateless tests do not maintain state between action executions, nor do they care about semantics of sequences of such actions. They may operate on the same object for more complex state transitions for the manipulated O_i.

It is essential to note the tight coupling of actions with their parameters and a corresponding distribution to walk different paths.

An example of a stateless test applied in Concurrency Runtime is the parallel_for stress where parallel_for is called with uniformly distributed numbers within the domain of parallel_for index type. Here the object is an instance of parallel_for class and the action is the () operator. Note that test iterations don't store state and do not remember the previous inputs to the parallel_for () operator.

ii) Scenario based / State full

Scenario based tests have a predefined structure on how to manipulate the object under test. The structure can be viewed as a set of Action Groups (AG) where each group manipulates the Internal State (IS) of the test which is passed to the next group.

$$AG = \left\{\left\{O_i, A_{ij}, IS\right\}\right\} i = 1,2, \dots, ObjectCount, j = 1,2, \dots, ActionCount$$

$$Scenario\ based = \{AG_k\}\ k = 1,2, \dots, GroupCount$$

Stateless tests can be seen as a specialization of scenario based where $k = 1$ and $IS = \{\}$. Scenarios end up targeting the semantics of sequences of actions.

An example of a scenario based test is a tree traversal scenario where nodes of a tree are traversed via parallel constructs of the Concurrency Runtime such as; parallel_for, parallel_invoke, task_group, etc... Here, IS is the tree data structure, O_i are the parallel constructs and A_{ij} are actions like scheduling a task, cancelling a parallel_for.

3.5.2. Stress Test Combiner Tests

Individual stress tests are not able to trigger race conditions that require more than one scenario running at the same time. It is important to have a combiner test that takes multiple individual stress tests and integrates them to compose more complex scenarios. Most likely there will be one combiner test for a group of related objects of the software under test. Combiner tests can also maintain a pool of related objects and pass them to the individual tests to have them operate collectively on same objects. This will result in more complex state transitions by reusing existing individual tests.

3.6. Runtime Randomizations

It is important to note that for lock free software such as the Concurrency Runtime, the threads can execute for their full quantum without blocking. Therefore even with randomization embedded into the tests there can be blind spots where the execution is not interrupted. In order to overcome this issue, runtime randomizations are introduced. The idea behind the Runtime Randomization is to introduce even more interleavings by randomizing the thread schedules of a given process.

4. CTM Implementation for Concurrency Runtime

In this section we will describe the implementation characteristics of the proposed CTM for the Concurrency Runtime.

4.1. Randomizer

Randomizer is implemented as a library that provides configurable randomization support to the stress tests. Each test has to define the random distribution type and parameters in a configuration file, that best models the expected customer usage. There are four different types of distributions supported:

i) Uniform distribution
This will generate numbers homogeneously in the range of $[Min, Max]$

ii) Gaussian distribution
This will generate numbers according to the Gaussian curve characterized by Mean and $\mathrm{Standard\ Deviation}$.

iii) Exponential distribution
This will fit an exponentially decreasing curve from Min to Max and generate random numbers accordingly.

iv) Weighted distribution
This will generate numbers from an $[\mathrm{Min_i}, \mathrm{Max_i}]$ interval homogenously where selecting an interval has the probability

$$\mathrm{P_i}\ , \mathrm{i} = 1,2, \dots, \mathrm{IntervalCount}$$

where

$$\sum_{i=1}^{\text{IntervalCount}} P_i = 1$$

Thus Weighted Distribution (WD) can be defined as:

$$\text{Interval}_i = \{\text{Min}_i, \text{Max}_i, P_i\}\ i = 1,2, \dots, \text{IntervalCount}$$

$$\text{WD} = \{\{\text{Interval}_i\}\}\ i = 1,2, \dots, \text{IntervalCount}$$

Note that weighted distribution can be used to approximate any random distribution:

Let $P(x)$ be the probability density function of a random variable x, where $x \in [\min, \max]$. The $[\min, \max]$ can be partitioned into $\{[\min_i, \max_i]\}$ such that for any $x_0, x_1 \in [\min_i, \max_i]$ $(\text{Eq. } 1)$ holds.

$$|P(x_0) - P(x_1)| < \text{Threshold} \qquad (\text{Eq. } 1)$$

P_i would be then defined as:

$$P_i = \int_{\min_i}^{\max_i} P(x)dx$$

Note that Threshold as defined in $(\text{Eq. } 1)$ determines the precision of the approximation; the smaller it is the better approximation is.

4.2. Individual Stress Test Structure

Individual stress tests are derived from a TestBase class for which the driver uses to initialize, start, cancel and clean-up the test. Each test implements the following:

Exported _cdecl APIs:

- TestBase* CreateInstance(): To create the stress test
- Void DestroyInstance(TestBase*): To destroy the stress test

Virtual methods:

- OnTestStart(): To initialize the stress test.
- TestMain(): To define what the test will do. This method is executed repeatedly until the test is canceled. It is expected that this method returns before the hang detection configuration parameter for the test. Otherwise test is considered as hung, the process state will be dumped and process will be exited.
- OnTestCancelled(): To handle test cancellation. Test is expected not to add more work and finish its tasks as quickly as possible upon cancellation.
- OnTestEnd(): To clean up test resources

4.3. Individual Stress Test Combiner Structure

Stress Test Combiner Structure is encapsulated in a class (MeltingPotBase) that each combiner derives from. The base class handles the common functionalities as follows:

- Loading a collection of individual stress tests at start up
- Scheduling a subset of tests simultaneously on multiple threads

- Collecting statistics about each test
- Reporting statistics upon failure

MeltingPotBase also has virtual methods that the combiner implementations can customize to do specific actions at individual stress load / unload, start and stop time.

4.3.1. The scheduling algorithm

MeltingPotBase implements a fair scheduler to decide which set of stress tests to execute in parallel and how long. Regardless of a test being short running or long running, the standard deviation of the total execution time of each test remains low. This is to prevent long running tasks from getting an uneven share of execution time over short running tests. The algorithm keeps track of average execution time of each test (AverageTestDuration) and creates a weighted distribution (WD) made up of intervals that represent the probability of selecting an individual test as follows:

$$\mathrm{WD} = \left\{\{\mathrm{Interval_i}\}\right\}\ \mathrm{i} = 1,2,\dots,\mathrm{TestCount}$$

$$\mathrm{Interval_i} = \{\mathrm{i}, \mathrm{i}, \mathrm{P_i}\}\ \mathrm{i} = 1,2,\dots,\mathrm{TestCount}$$

where

$$\mathrm{P_i} = \frac{\frac{1}{\mathrm{AverageTestDuration_i}}}{\sum_{\mathrm{j=1}}^{\mathrm{TestCount}} \frac{1}{\mathrm{AverageTestDuration_j}}}\ \mathrm{i} = 1,2,\dots,\mathrm{TestCount}$$

The algorithm will select N tests by generating N numbers from the WD where each number corresponds to the i^{th} test. The goal is to have all tests converge to the same execution time. Therefore for a faster convergence the algorithm runs the selected tests $\mathrm{C_i}$ number of times where:

$$\mathrm{C_i} = \frac{\max(\mathrm{AverageTestDuration_j})}{\mathrm{AverageTestDuration_i}}$$

Here are the algorithm steps:

- Create a weighted distribution (WD) inversely proportional to average run duration of each test
- Generate N unique numbers from the WD that indicates the N tests to be executed
- Attach each test to a thread
- Execute N tests in N threads C_i number of times
- Update statistics
- If cancelled exit otherwise continue from 1st step

4.4. Stress Test Driver

The implementation of the stress test driver is generic for all software systems under test. It implements the capabilities outlined under 3.3 Stress Test Driver.

4.5. Stress Tests

4.5.1. Individual Stress Tests

The Concurrency Runtime stress tests are composed of 27 individual tests where 25 of them can be combined with others. All tests use random distributions (there are 145 distributions in total). The test scope, count, and functionality is designed after decomposing the runtime into components,

understanding each component, their relation with each other together with the scenarios that could trigger interesting race conditions. Understanding functional specifications, design documents, source code and collaborating with developers are good practices to follow for a better test design.

4.5.2. Stress Test Combiner Tests

There is only one STCS implementation for the Concurrency Runtime implementation of the model which is called the Melting Pot. Melting Pot targets integration around the scheduler which is the most critical component of the runtime. It maintains a pool of schedulers and populates individual tests with an instance of a scheduler from the pool.

4.6. Runtime Randomizations

The runtime randomization (as described in section 3.6) applied for the Concurrency Runtime testing is called the Thread Randomizer. The randomizer will basically sleep for a configurable duration, suspend a number of target test execution threads, resume them and then go back to sleep. The number of threads to suspend and the duration of the sleep follow a random distribution and can be configured. This configuration enables different interleavings without modifying test behavior. Thread Randomizer is a simple external process that randomizes the thread scheduling without any modification to SUT or the tests. The idea is similar to [4,8] where instead, we use a uniform distribution to decide injection points.

It can be shown that the probabilistic guarantee of finding a bug of depth one is in the order of $O\left(\frac{1}{N}\right)$ where N (N>1) is the number of threads in the process (see Appendix I – Thread Randomizer bug finding probability proof).

5. Results

We will compare the results of applying concurrency testing model with other methodologies used for testing the runtime. In addition, the bugs found by our methodology will be examined to describe the effectiveness of techniques used. Finally a comparison with similar concurrency testing methodologies will be provided.

It is also important to note that the Concurrency Runtime team has no reported Watson bug (customer reported crash database) which is another indication of the effectiveness of reliability testing done.

5.1. Survey of Bugs Found

5.1.1. Overall Efficiency of Bug Finding Activities

We will lay out the efficiency of the stress testing by ranking it among other bug finding activities. The test methodologies that Concurrency Runtime has applied are as follows:

- Feature Testing (aka Functional Testing)
  Functional tests implemented by the test team
- Stress
  Tests that are implemented by using the methodology described in this paper
- Adhoc Testing
  End to end scenario based tests
- Code Reviews
- Unit Testing
  Functional tests implemented by the developer team

- Performance Testing
- Other
  Other buckets like bugs coming from Microsoft internal usage, security, etc…

The number of bugs for each methodology can be seen in Figure 3.

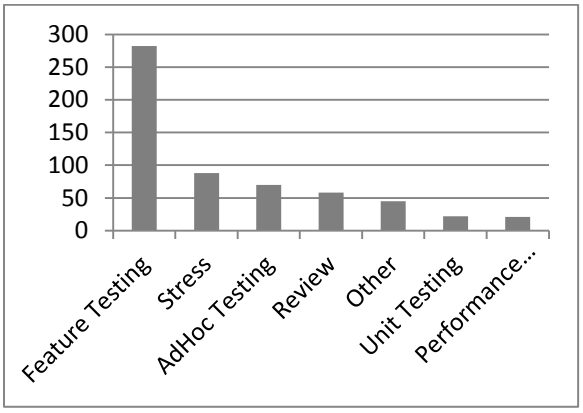


Figure 3 - Number of bugs for per testing methodology

One way to look at efficiency is the bugs found per lines of code written. We will evaluate the efficiency of top two bug finding methodologies; feature testing and stress.

Table 1 - Evaluating top two bug finding activity efficiencies

Bug Finding Methodology	Number of Bugs Found	Lines of code
Feature Testing	282	163552
Stress	88	63751

Data in Table 1 reveals the efficiency per 1000 lines of code as:

Efficiency of feature testing: 1.72
Efficiency of stress testing: 1.38

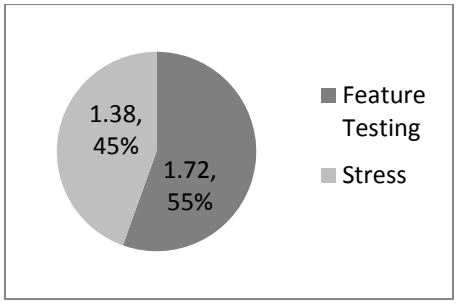


Figure 4 – Efficiency of top two bug finding methodologies

This indicates that even though the number of bugs found with feature testing is more than triple the number of bugs found with stress testing, the efficiency of the methodologies are comparable.

It is important to note that the bug types that are found by the stress tests are much more complex in nature with respect to reproducibility which we have not taken into consideration in evaluating the

efficiency. Another very critical point to make here is that the stress tests are typically developed after the functional tests have verified that the core testing is done.

5.1.2. Type of Bugs Found by Stress Tests

The analysis of bugs found can be another factor in evaluating the concurrency testing methodology. The results for the Concurrency Runtime experience can be found in Figure 5.

Access violations are a classic symptom of harmful race conditions. Live-locks and deadlocks are classic symptoms of incorrect synchronization and scheduling. The developer and test asserts1 are a symptom of incorrect state transition. By noting the variety of bugs that Stress finds, we can also assert its effectiveness.

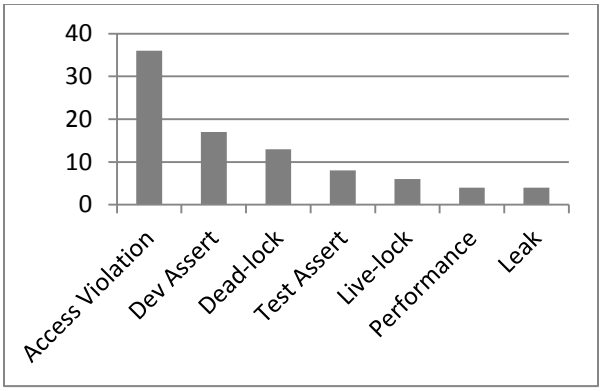


Figure 5 – Type of bugs found by stress tests

The performance and leak buckets also appear in the analysis. Even though stress does not target performance verification, performance characteristics of the software under test may end up making the test take too long under heavy load and not complete in a predefined amount of time. Such cases are detected as hangs by the stress framework but further analysis yields to underlying performance issue. On the other hand, certain leak bugs are exposed due to race conditions or after running a long duration which is very hard to catch on environments without targeting stress testing.

5.1.3. Effectiveness of Stress Tests

We have made a distinction between individual stress tests and the melting pot that combines individual tests. We were looking at melting pot as the strategy to traverse various race conditions and exercise various scenarios that are not possible by running the individual tests. The following graph is a validation of that strategy that melting pot on its own has detected almost 1/3 of total stress bugs (The analysis is based on the 'repro steps' field in the bug database. If repro steps refer to the melting pot, the bug is marked as detected by melting pot). This is even more important since the individual tests were first stabilized outside of the melting pot environment. Only after the individual test runs successfully is it added to the melting pot mix.

The second biggest effective test is the tree traversal test case which is a unique combination of stressing all parallel constructs of the Concurrency Runtime under a tree data structure traversal user scenario. Tree traversal also involves verification which makes it even more powerful bug detector. For a detailed effectiveness of stress tests see Figure 6. Melting pot and tree traversal indicates that composition of tests or features in order to exercise more complex scenarios has a greater chance of detecting bugs.

[1] There is a natural tension between using random distributions and being able to inject verifications. The more randomness is introduced the less verification can be done.

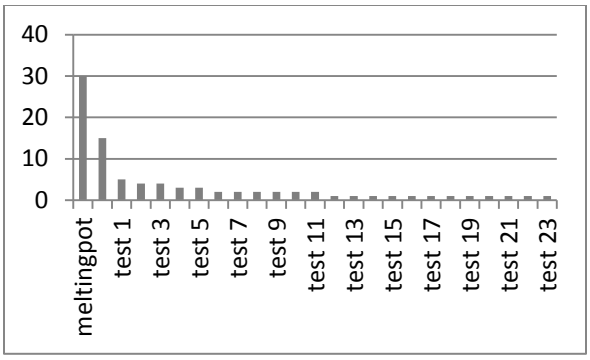


Figure 6 – Effectiveness of stress tests

5.2. Comparison to Other Concurrency Testing Methodologies

5.2.1. Repetitive Functional Test Execution

One way for finding concurrency bugs is reusing functional verification tests and running the tests over and over until cancelled. Such approaches also have capabilities to run tests from multiple threads. We applied this approach to existing functional tests of the Concurrency Runtime. Only the tests that were capable of running with other tests at the same time were harnessed. Tests were selected randomly and ran in multiple threads. Even though the implementation of this approach is easy due to excessive re-use of existing tests, the number of bugs found remain as two which suggests that the methodology is not enough on its own.

The low detection ratio can be explained by two features missing in such an approach:

i) Limited randomization
The functional tests are deterministic in nature; there is minimal tolerance to flakiness in functional tests. Thus the only way to exercise different interleavings is to rely on the noise related with the system load or the OS thread scheduler.

ii) Limited shared runtime state
The individual functional tests do not know about other tests in their environment. They do not share state explicitly. They act on their own instances of objects and also clean-up resources every time the test completes. This prevents more complex states from being reached as the software under test resets every time a new test starts.

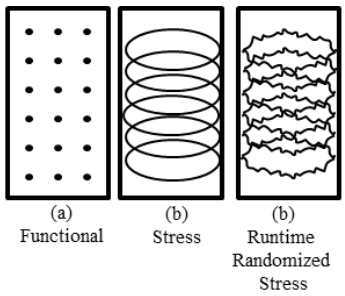


Figure 7 – Depicting functional vs. stress test coverage

In order to visualize the difference between repetitive functional test execution and stress, let us assume that Concurrency Runtime usage scenario space is a two dimensional space[2]. We interpret the coverage of a functional test within this space as a dot (Figure 7a) because of its deterministic input and execution logic. On the other hand, due to the introduction of well-defined random distributions into the test code, we interpret the coverage of stress tests as a collection of regular dots (Figure 7b) and as a result of thread randomization, we interpret runtime randomized stress test coverage as a collection of irregular dots (Figure 7c).

5.2.2. Systematic Concurrency Testing - CHESS

CHESS [6] is a tool which systematically explores the schedules that a concurrency program can exercise. Running a test, which does not introduce randomizations by itself, CHESS will control the scheduling of threads and will exercise a different path each time the test executes. CHESS will also record the scheduling decisions made so that if a specific schedule hits a bug, it can be reproduced. Due to the exponential growth capacity of the search space, CHESS will limit its systematical search with a bound on the number of preemptions it makes on the threads [6].

For software as complex as the Concurrency Runtime, we have observed that systematic approach of CHESS is infeasible due to the length of time to complete testing. Thus, a stochastic walk in our framework provided more practical coverage and proved to be a better tool to predict quality.

5.2.3. Randomizing Thread Schedules - Cuzz

Cuzz [5] is an implementation of runtime randomization introduced in paragraph 3.6. Cuzz introduces the concept of bug depth as the minimum number of scheduling constraints needed to expose a bug. Cuzz will randomize thread schedules and guarantee to find a bug of depth ‘d’ for a test having ‘k’ scheduling points and n threads as 1 / (n x kd-1) [5]. Here the scheduling points are instructions for which thread preemption has a potential of exposing a bug. Examples of such scheduling points are interlocked operations, critical section operations, etc…

Cuzz and Thread Randomizer have similar goals in that both of them aim to randomize the thread executions so that race conditions are exercised. Cuzz will do this by injecting delays at interesting scheduling points. On the other hand, Thread Randomizer will randomly suspend/resume any thread with no bias to scheduling points. From this perspective, Cuzz has a better probability of finding a bug. On the other hand, Cuzz can have blind spots.

After the public release of the Concurrency Runtime, we have been running stress with Cuzz. We have not yet discovered a bug, which is another indication that the concurrency testing methodology described in this paper is firm.

6. Conclusion

In this paper we have described a framework to cope with the challenges of testing concurrent software. Our model is composed of data driven random distribution support, individual stress tests with parameters coming from random distributions, integration of such individual tests for more complex state transitions and runtime randomizations for finer grain interleavings. By providing real data from the bug database of the Concurrency Runtime and comparing our methodology with other concurrency testing methodologies, we demonstrated the effectiveness of the methodology.

We believe that applying data driven random distributions into test executions, combining existing tests on shared state to explore even more complex scenarios, and introducing thread scheduling randomizations in order to probabilistically guarantee finding concurrency bugs arising from different interleavings is an effective way of testing highly concurrent software.

[2] The assumption is for visualization purposes of our interpretation.

7. Appendix I – Thread Randomizer bug finding probability proof

Let us define the bug depth as the number of ordering constraints needed to be met to end up with a given bug. This is essentially the definition given in [5]. For a simple scenario let us visualize this as follows. Note that two threads are enough to have this bug occur.

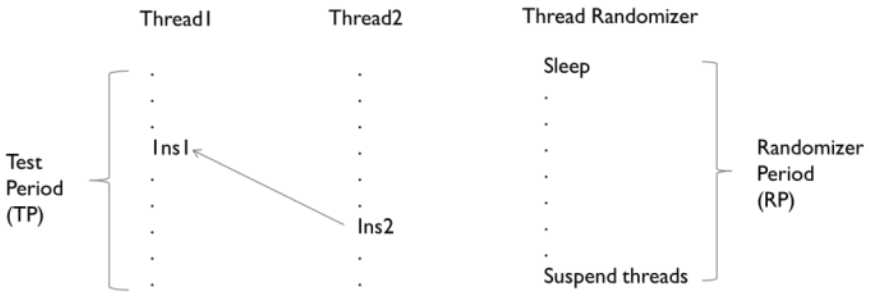


Figure 8 - Thread Randomizer typical runtime behavior for a test with two threads

We will model the test doing the following in a loop:

- Create and start two threads
- Wait for the threads to end

Note that threads are free to do whatever action they want to perform. We will define each execution of the loop as test iteration and the duration of each test iteration as TP. We will denote MINTP as the minimum duration of any possible iteration and MAXTP as the maximum duration of any possible iteration.

Meanwhile Thread Randomizer will do the following in a loop:

- Sleep for its period (RP)
- Wake up
- Suspend threads for Suspend Duration (SD) amount of time

We will define each execution of this loop as randomizer iteration.

For a bug depth of one we will assume that the Thread Randomizer will suspend one thread at a time. Since Thread Randomizer is capable of getting its parameter values from random distributions, we will define RP values coming from a uniform distribution on interval $[\mathrm{MINRP}, \mathrm{MAXRP}]$ and SD values coming from a uniform distribution on interval $[\mathrm{MINSD}, \mathrm{MAXSD}]$.

Note that the bug will occur if Ins1 (instruction 1) happens before Ins2 (instruction 2) (See Figure 8). For this bug to happen under thread randomization:

i- Thread1 needs to be suspended before executing Ins1
ii- Suspension must be long enough for Thread2 to execute Ins2

Since Thread Randomizer is configured to suspend at minimum MINSDms we will assume that Thread Randomizer cannot detect bugs if the duration between Ins1 and Ins2 is longer than MINSD. If MINSD is not enough to hit the mentioned bug then the Thread Randomizer's configuration can be modified to increase the MINSD. Thus we will assume that (ii) is met by definition.

Let us define IPP as the number of instructions that can be executed on the test machine in TPMAXms. Given that Thread Randomizer is unbiased (it interrupts the test process after sleeping randomly between $[\mathrm{MINRP}, \mathrm{MAXRP}]$) each test process instruction has equal chance of being interrupted. Then the probability of inserting a sleep before an instruction at k^{th} location would be $\mathrm{P} = \frac{\mathrm{k}}{\mathrm{IPP}}$. Let $\mathrm{k1}$ be the index of

Ins1 with respect to Thread1's executed instruction set. If there are N threads, then probability of selecting Thread1 and injecting before Ins1 would be:

$$P(Thread1) = \frac{k1}{IPP} \times \frac{1}{N}$$

Since Thread Randomizer can miss some test iterations completely while sleeping, we need to take into account the probability of selecting test iteration i. The probability of hitting test iteration within Thread Randomize iteration is the ratio of test iteration to randomizer iteration.

$$P(TestIteration_i) = \frac{TP}{RP + SD}$$

A lower bound would be:

$$P(TestIteration_i) > \frac{MINTP}{MAXRP + MAXSD} \quad (Eq.2)$$

Note that the following must be satisfied to have $P(\text{TestIteration}_i) < 1$:

$$MAXRP + MAXSD \geq MINTP$$

Then probability of finding the bug in test iteration i would be:

$$P(bug) = P(Thread1) \times P(TestIteration_i)$$

$$= \left(\frac{k1}{IPP} \times \frac{1}{N}\right) \times P(TestIteration_i) \qquad (Eq.3)$$

Lemma 1: Given the fact RP and SD are derived from a uniform random distribution, $k1$ can be estimated as $\frac{IPP+1}{2}$.

Proof: Since RP and SD values are uniformly distributed, the rank of $k1$ with respect to Thread Randomizer iteration will be a uniformly distributed between 1 and IPP. Possible values for $k1$ become:

$$k1 \, \epsilon \, K = \{1, 2, \dots, IPP\}$$

where each value has probability of $\frac{1}{IPP}$. Let us visualize this as shown in Figure 9.

Thus the estimated value of $k1$ becomes:

$$E(k1) = \sum_{k \in K} \frac{1}{IPP} \times k = \frac{1}{IPP} \times (1 + 2 + \cdots + IPP) = IPP \times \frac{IPP + 1}{2IPP} = \frac{IPP + 1}{2}$$

Replacing $E(k1)$ in Eq.3, we have:

$$P(bug) = \left(\frac{\frac{IPP + 1}{2}}{IPP} \times \frac{1}{N}\right) \times \; P(TestIteration_i)$$

$$= \left(\frac{IPP + 1}{2NIPP}\right) \times \; P(TestIteration_i)$$

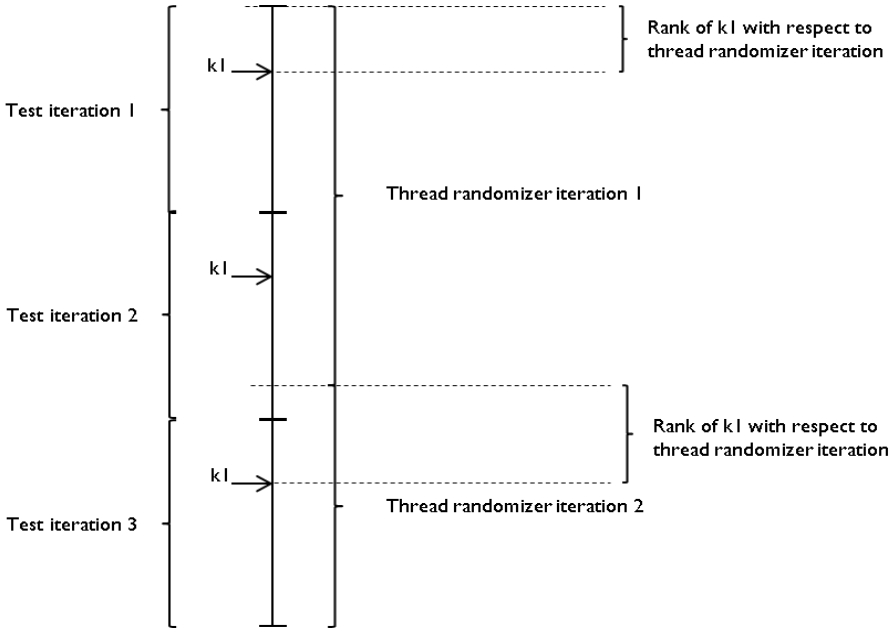


Figure 9 - Runtime visualization of thread randomization

Since we are interested in a lower bound for the probability we can lower it by dropping the positive terms. Thus we have:

$$P(bug) > \left(\frac{IPP}{2NIPP}\right) \times P(TestIterationi)$$

$$= \frac{1}{2N} \times \frac{TP}{RP + SD} \quad (Eq.4)$$

Replacing Eq.2 in Eq.4:

$$> \frac{1}{2N} \times \frac{MINTP}{MAXRP + MAXSD}$$

Thus the probability of detecting the bug of depth one in a process having N threads is in the order of $O(\frac{1}{N})$.

This indicates that running the test for $\frac{1}{P(bug)}$ times will reveal the bug.

References

[1] Narciso Cerpa and June M. Verner. “Why did your project fail?” Communications of the ACM, December 2009, Volume 52 Issue 12.

[2] Gerard J. Holzmann. “The logic of bugs.” SIGSOFT '02/FSE-10: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, November 2002.

[3] Concurrency Runtime, http://msdn.microsoft.com/en-us/library/dd504870(VS.100).aspx

[4] Y. Ben-Asher, Y. Eytani, E. Farchi, S. Ur. “Producing scheduling that causes concurrent programs to fail.” PADTAD '06: Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging, July 2006.

[5] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, Santosh Nagarakatte. “A randomized scheduler with probabilistic guarantees of finding bugs.” ASPLOS '10: Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems, March 2010.

[6] Madan Musuvathi. “Systematic concurrency testing using CHESS.” PADTAD '08: Proceedings of the 6th workshop on Parallel and distributed systems: testing, analysis, and debugging, July 2008.

[7] Rahul V. Patil, Boby George. “Tools and Techniques to Identify Concurrency Issues.” MSDN Magazine, June 2008. http://msdn.microsoft.com/en-us/magazine/cc546569.aspx

[8] Yarden Nir-Buchbinder, Shmuel Ur. “Multithreaded unit testing with ConTest.” http://www.ibm.com/developerworks/java/library/j-contest.html

[9] Li Jiang, Armin Eberlein. “An analysis of the history of classical software development and agile development.” Proceedings of the 2009 IEEE International Conference on Systems, Man, and Cybernetics. October 2009, Page(s): 3733 - 3738

Acknowledgements

We would like to thank Mohamed Ameen Ibrahim for sharing his bug trend analysis on the Concurrency Runtime, Steve Gates for providing investigation results of applying CHESS on the Concurrency Runtime as well as his review on the paper, Roy Tan for his review, the Concurrency Runtime Test Team for implementing tests on top of the model and the Concurrency Runtime Developer Team for their support in designing test cases.

Using Silverlight PivotViewer to make sense of the chaos

Max Slade, Frederick Fourie, Melinda Minch, Ritchie Hughes

maxs@microsoft.com

Abstract

Today's methods of organizing and searching information cannot keep pace with the exponential rate of growth. This information overload presents an opportunity to expose new value in the aggregate, where the data interactions provide greater ability to act on previously unexplored insights.

PivotViewer will change the way people perceive and explore the information that surrounds them by visualizing information in new ways, exposing hidden relationships, and making it easier to act on these newly discovered insights. This is possible because PivotViewer can simultaneously show thousands of items on screen at once while allowing you to effortlessly browse around the information by selecting properties that you want to explore. You are also able to zoom into a single item and get more information and then pivot on any property that the item has.

In an industry which generates vast quantities of data in the form of defect reports, test results, code coverage, and other metrics, Testers can use PivotViewer to quickly find trends and anomalies, visualize expected trends for consumption by decision makers, and focus our limited resources on the areas that matter most. We expect that PivotViewer can be used in ways that we haven't yet imagined.

Biographies

Max Slade is a Principal Test Manager at Microsoft's Live Labs. Max has worked at Microsoft in the test discipline since 1996. He has led the test teams for PivotViewer, Live Labs, Desktop Search, and NetGen. Max has a bachelor degree in Electronics Technology from Chapman University

Frederick Fourie has been developing software at Microsoft for nearly ten years on products ranging from Windows to Robotics. Frederick has a degree in Computer Science and Applied Mathematics from Rhodes University (South Africa) and is currently working on Microsoft Robotics Studio.

Ritchie Hughes is a software design engineer in test at Microsoft's Live Labs, located in downtown Bellevue, Washington. Ritchie's career at Microsoft started with building tools and test automation for the original Windows Mobile developer platforms, before moving on to an incubation project that become Microsoft Surface. Ritchie has a bachelor degree in Computer Science from the University of Cambridge.

Melinda Minch is currently a tester at Getty Images in Seattle, Washington, where she works on the main website. Melinda has an M.S. in Computer Science from Case Western Reserve University and a long-standing interest in multivariate data visualization.

1. Introduction

As testers, one of our main functions is to measure things that give visibility into the quality of the software we're building. We measure defect reports, functionality, performance, stability, stress, load, localizability, etc. and in doing so generate staggering amounts of information. There is so much information generated while testing a typical software project that it's difficult to derive the full value from all that is produced. We settle for generating charts, graphs, or tabular information that are adequate at answering directed questions. However, there is a wealth of untapped information trapped in those formats that are not easily viewed in aggregate. There are veins of information gold hidden in those repositories.

Now it is possible to prospect with PivotViewer, a powerful tool that allows you to view your test data in a way that's powerful, informative, and fun. With PivotViewer, you can visualize, explore, and understand large sets of data; we need not be in a state of information overload. Now we find that the whole is greater than the sum of the parts. PivotViewer makes it possible to truly understand the information. It is possible with PivotViewer to move from many items to one item, from many items to many related items, or from one to many related items.

PivotViewer is a Silverlight Control that can be used to view the data that you generate while testing your product. By using tools provided by the PivotViewer team, you can translate your data into the two formats that PivotViewer requires – CXML (an XML schema), and Deep Zoom images. Then using traditional web hosting or the Just in Time server that we provide sample code for, you can make your data available in PivotViewer. Detailed information about Deep Zoom can be found here: http://www.silverlight.net/learn/quickstarts/deepzoom/ .

PivotViewer has been used by several Fortune 500 companies to gain insights from Business Intelligence information that were previously hidden, required an expert to detect, or were difficult to explain to decision makers via tables or spreadsheets.

Live Labs at Microsoft released the PivotViewer experience with two different technologies. Last November the Pivot WPF client was released as an experiment. We expected that once we released the Pivot experience to the world, we'd see uses that we hadn't imagined ourselves. This happened in a big way, primarily around how different companies used it for Business Intelligence. Corporations found that putting the data that they use to judge business success into Pivot was amazingly powerful. The success of the WPF client led to many requests for a way to have the Pivot experience in any browser and on any Operating System. Our answer came at the end of June this year when we shipped the Silverlight PivotViewer control.

In this paper I will describe the application of PivotViewer to the test discipline by walking through three example sets of information: Bugs, test run results, and code coverage. Since PivotViewer is a data visualization tool, I'll be providing screenshots and a narrative to demonstrate the mechanics of using it. However, no matter how well I convey ideas in this paper, it will not be a substitute for seeing it yourself as the information architecture relies on animations during transitions of views on the data. The key points in this paper can be significantly enhanced by trying it yourself or seeing one of the videos of PivotViewer in action available at www.getPivot.com .

Following the examples, I will describe how to put your own data into a PivotViewer collection and provide a high level description of the architecture of the Viewer and hosted data. The term "collection" has special meaning that will be described in detail. The most basic definition of a collection is the full set of

items that PivotViewer can display at once without any filters applied. A collection is the set of items and their metadata described in an XML file and associated imagery stored in a Deep Zoom collection.

In conclusion, I will have a call to action asking how you imagine PivotViewer can be used by you and your organization or by others.

2. Bugs

In the following example I'll walk through an example of a collection of Product Studio data created to explore our defect tracking. Product Studio is a bug tracking database used extensively at Microsoft. I'll describe what the collection contains, how we selected the properties to include, and how we created the visuals for individual bugs. The figures below will show the entire data set (the forest), various slices of the set by *facets* (trees), and individual bugs (leaves). Facets are properties that the items within the collection share. The example below also walks through how I explored the data. There are thousands of ways the data could be viewed. I'm going to show a selection that I hope will inspire others to see ways that PivotViewer could be used with their own data.

With PivotViewer, we can expose the important fields in the bug database as parameters (facet categories plus facets) in the filter pane – the left side of the PivotViewer (Figure 1a). We chose Status, Priority, Opened By, Resolved By, Closed By, Issue Type, Triage, Resolution, Node Name, Opened Date, Closed Date, and Days Active. All but the last facet category are pulled directly from the database. The Days Active facet category is calculated by doing a simple calculation (Date Resolved – Date Opened). This was an important factor for the PivotViewer development process. Minimizing the time between code creation and defect fixing reduced the cost of the developer having to context switch from new tasks to older tasks giving a higher velocity of feature development (Figure 1b).

Figure 1a. Six months of bugs from one of our projects. The left pane is referred to as the Facet pane. For example, Priority is a Facet Category and 0, 1, 2, 3, and 4 are facets. The counts for each facet are displayed just to the right of the facet. The current view shows all bugs opened during this time period. The colors of the items in view represent the Status of the bug. Red for Active, green for Closed, and

yellow for Resolved. Active means the bug hasn't been fixed yet. Closed means the bug has been fixed and verified by a tester in the product. Resolved means the developer believes the bug to be fixed but not yet verified by a tester.

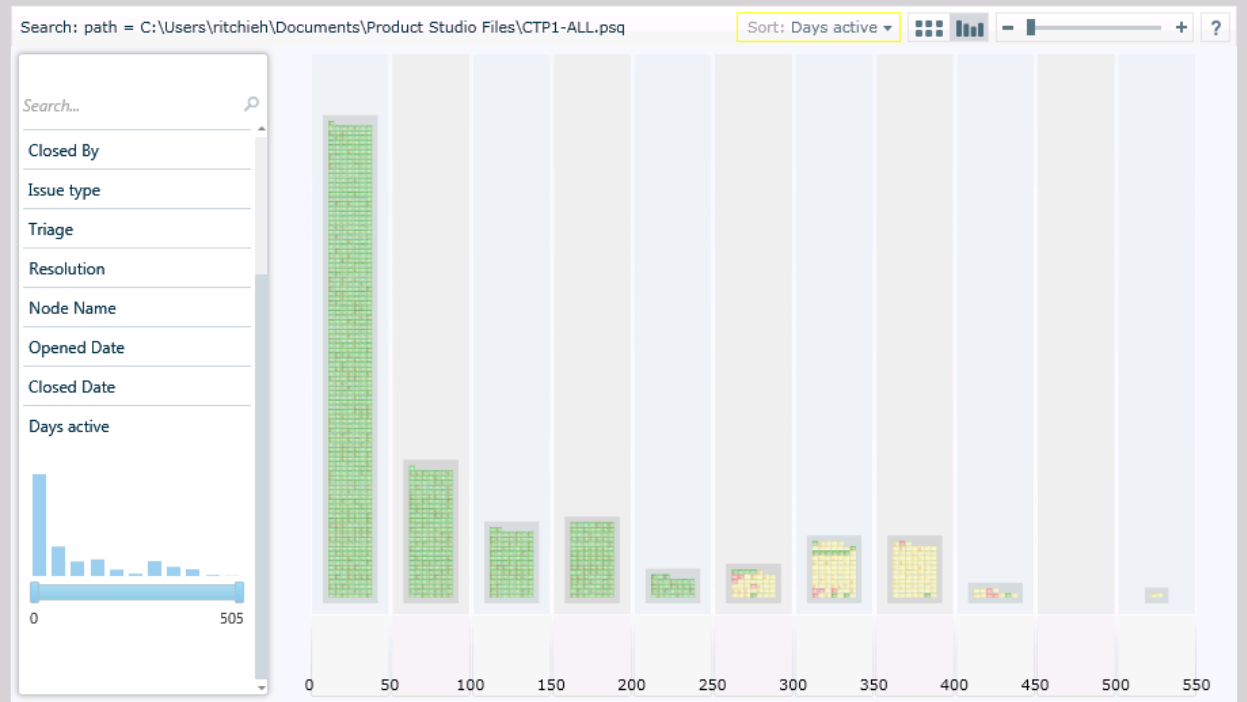


Figure 1b. Distribution of bugs over an entire ship cycle showing number of days bugs were active before being resolved.

In general, to create a great collection we've found that seven to ten facet categories are the right number. Usually fewer facet categories is a shallow set of data and more than ten appears too complicated. However, with a database as rich in metadata as a bug database and considering that the audience using the collection will be familiar with these fields we made an exception by including more than ten. Another best practice is to order the facet categories in order of importance. With our bug database collection we included the important field values as facet categories in an order that made finding the most commonly used fields easy.

Fields can be shown in the pane on the right (info pane) once zoomed in on a single bug (Figure 2) and each one of those is a clickable link to pivot to see all bugs sorted by that value. In this collection, the info pane is only providing another way to transition to other facet categories. In other collections, there is value in exposing facets that aren't important enough to include in the Filter pane (left pane) yet are interesting to expose while looking at a single item.

Once you've decided what metadata to include for each item, the next step is to create a Deep Zoom image for a visual representation of a bug (Figure 2). We have used a declarative language that allows us to create visuals at query time that map the data to a visual representation. We have used color, text, and

shape to represent our data. We've found that a dominant background color tied to Status (Fixed, Resolved, Closed, etc), the Title at the top of the bug, a secondary color for Priority including the number in one box and a second box containing the Resolution work very well in conveying information in this context.

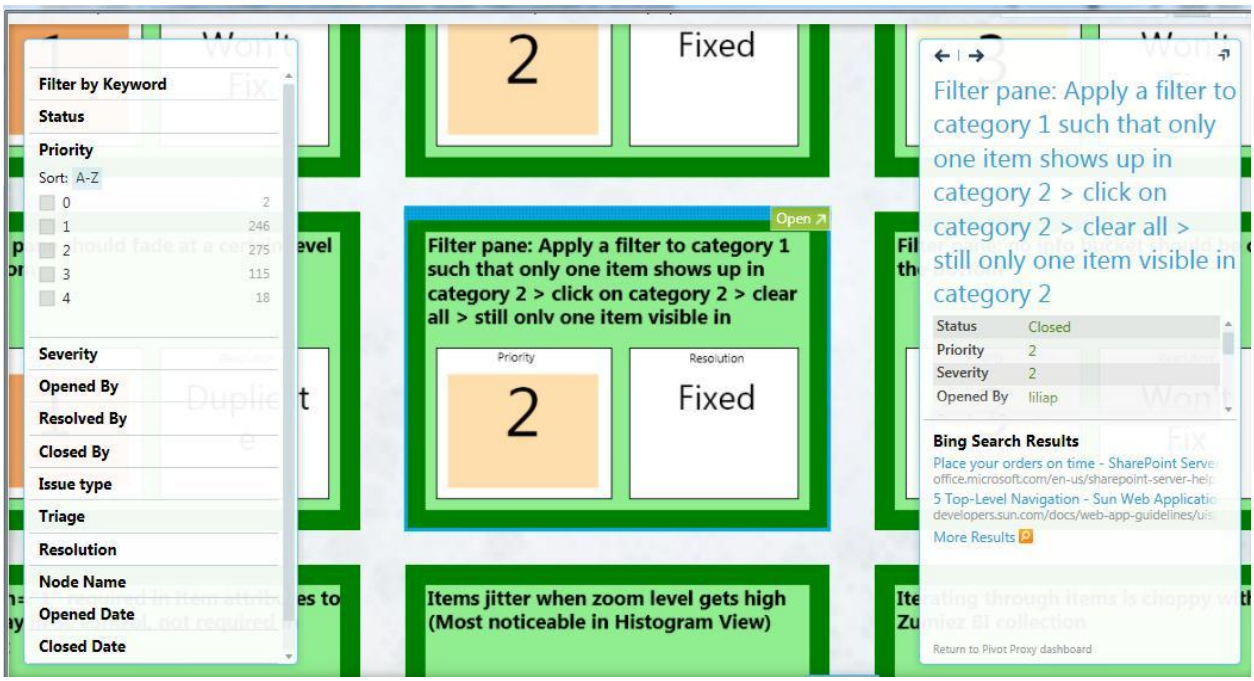


Figure 2. *Here is an example of zooming in on a particular bug – note that more detail emerges in the bug itself and the info pane on the right as you select an item. Items can be zoomed into by clicking on them or by using the mouse wheel to zoom. Once an item is selected it is possible to iterate through the collection by using the arrow keys.*

Since it's not possible to read the text when viewing the entire collection of items at once, we have a transition to just the dominant color when zoomed out (Figures 1 and 3).

The real power of PivotViewer comes from being able to easily swim around in this collection of bugs. Instead of relying on pre-authored queries, charts, or graphs, one can visualize this typically non-visual data in many interesting ways. Figure 3 is a view that shows several nodes in our bug database using histogram view. Nodes are a structure defined in the Product Studio database that is a hierarchical definition of how we categorize bugs. Consider nodes trees within the forest by way of analogy. This is powerful because with this one view I can gather quite a bit of information (See Figure 3). I can see the relative number of bugs in each node, the resolution counts in the filter pane on the left side, and the status as indicated by the color of the items. I can also easily learn about the distribution of any of the other filter pane groups by selecting them – the information is now relevant to the filtered set of information, not the entire collection.

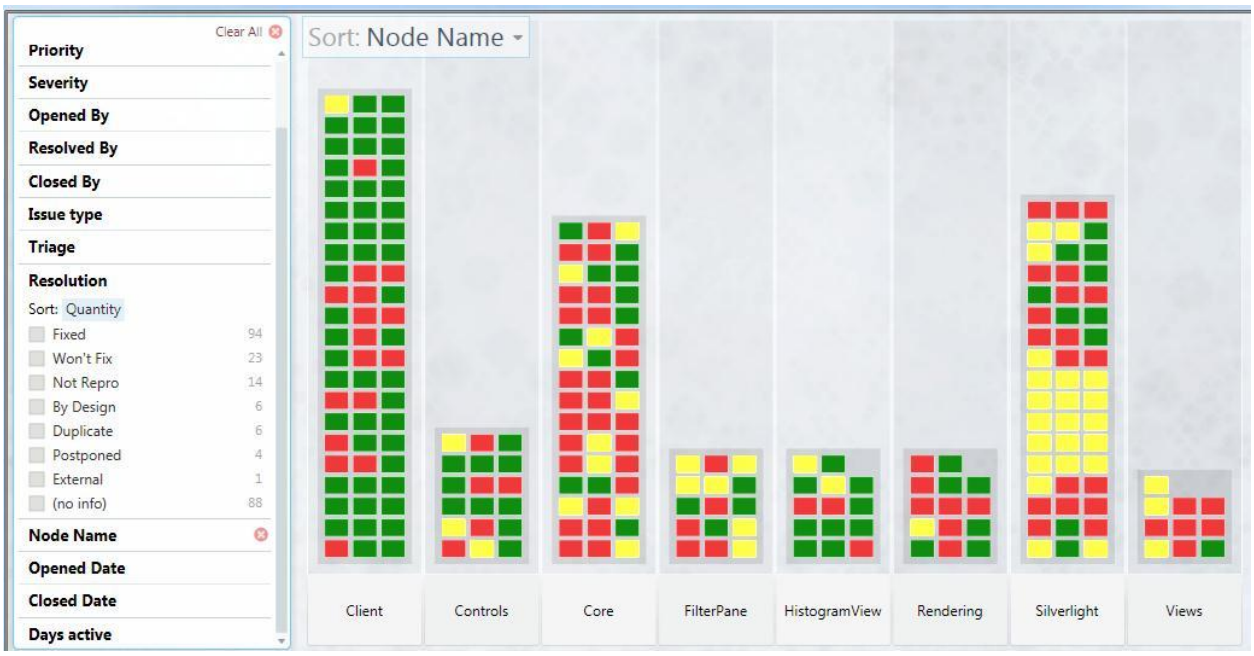


Figure 3. Histogram view of various nodes in a Product Studio bug database. In this view I've selected a number of functional and technical areas (nodes in the Product Studio database) by selecting them in the Filter pane (left side). The view is then sorted by Node Name (see "Sort: Node name" above the view). This particular view showing a large number of bugs in "Core" is interesting since we want core components to be solid so the features built on top are on a more stable code base.

PivotViewer can be used to easily find missing information. For example, when we first put our bug database information into PivotViewer, I quickly learned that some of the information we want people to enter was incomplete. The Severity field wasn't always being given a value (Figure 4) because of all of the bugs that had (no info) for that value. If Severity was being used by the team to improve prioritizing work or as a way of providing better visibility into the quality of the project, then this would be a useful discovery and could be corrected with the team.

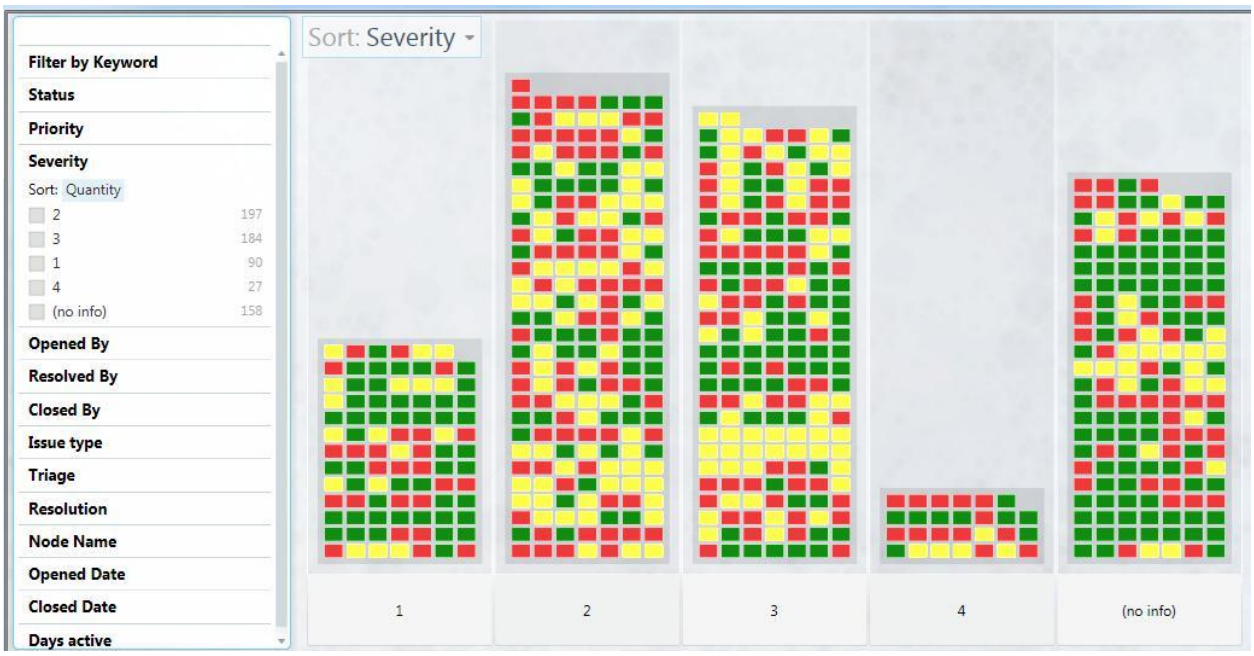


Figure 4 – Note bugs on the right side of the view with "(no info)". The color coding in this view for Status can be ignored.

Next, I graphed the bugs over time with the Opened Date field. This revealed the trend that I expected given the releases we had in that timeframe (See Figure 5a and 5b with defect issues graphed over time). This is valuable for a release manager to verify trends that are expected in an intuitive visual way. These views also facilitate showing others on the team the quality of the project in an intuitive easily understood way. The combination of the filter pane and the histogram with the visual display of data works with both the left and right brain. This combination brings information understanding that has not yet been available in a generalized way before. One can find better visual representations of information for any give dataset but PivotViewer provides a consistent and intuitive way to see multi-dimensional data representations for any data.

For example, in Figure 5b I can see what days of the week tend to have the most bugs opened in a given set of functional areas. The color coding tells me the status which may be important if a milestone is approaching. More importantly, this view can show the trend of bugs being opened over time for a particular feature area. Instead of having a holistic product view, I can drill into a feature area or set of features to gauge the health of this area. Besides getting a deeper understanding of the quality of a feature area, I can share this view with others and it is easily understood due to the intuitive graphic display. Each view state has a unique URL so it's easy to share any interesting views you find with others on the team.

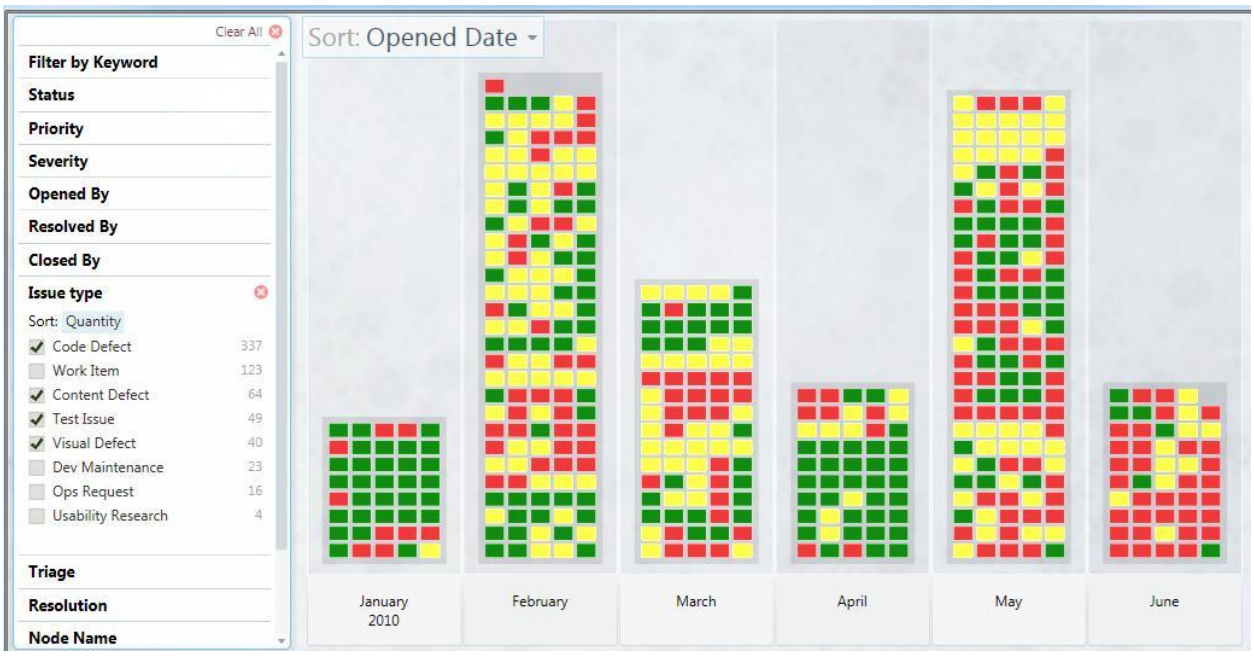


Figure 5a – Defects displayed over six month period

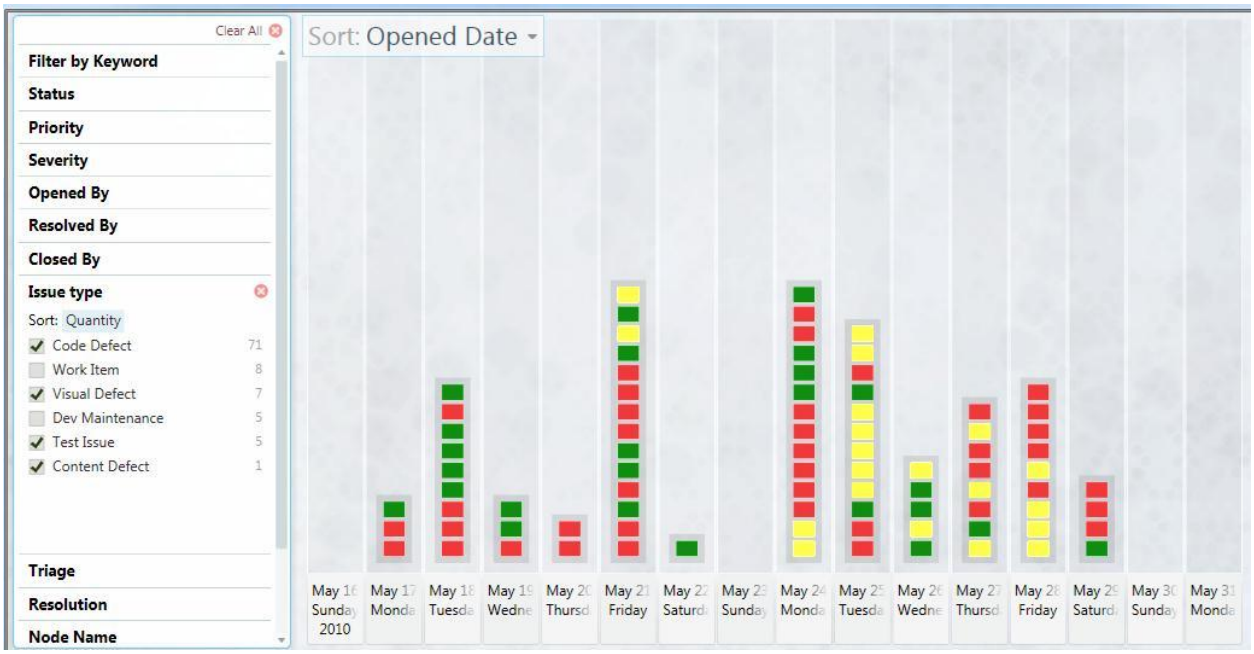


Figure 5b – Defects displayed over a two week period

Since it's easy to browse around the information I discovered some information that I hadn't known to look for. I realized that we have fixed bugs that were not approved by Triage. This informed me of a flaw in our process and prompted me to follow up on why we were working on bugs that may not have been required for release (see Figure 6).

Figure 6 – Bugs that were fixed, yet not Triage approved. Every bug in this view has been resolved fixed and the Triage counts show that 21 bugs are marked "Needs Triage" and 10 are marked "Investigate". This is a breach of our processes and a result of seeing these bugs lead me to find out why this occurred and find a way to prevent this in the future.

One thing that is difficult to explain in a paper with static screen shots like this is the powerful and fun transitions that occur when slicing and dicing the information with the filter pane or by clicking on columns. The animations that occur are powerful because they give the user a sense of context between a subset of the information and whole of the data, or from one subset to another. We aimed for a frame rate of 30-60 frames per second which gives a really smooth appearance. The fun part is that it's simply amazing to have the power to arrange information so easily and in such a pleasing way. You will find that it's fun to explore information that you cannot currently imagine is fun to peruse today.

I've given a single data source as an example but PivotViewer is designed to be a ubiquitous information visualization tool. We've also created collections with test run results and code coverage results.

3. Test run results

The collection found below in Figure 7 and Figure 8 contains around ten thousand test run results. This collection is interesting because it can decrease time spent debugging test failures. With PivotViewer, you can easily see common elements between the failures which will help you quickly determine where the root problem lies. In the test run data for Seadragon, PivoViewer was used to see test trends over time and on different hardware. It was now possible to see the commonalities in failures over multiple runs. This was particularly interesting because Seadragon did multiple runs with different hardware and it was much easier to spot the root cause within a pattern using PivotViewer. One helpful visual that was added to the visuals was the set the tics (sparkline) next to the Pass/Fail text in the item to indicate if this test has been flakey in the past or has just failed for the first time in a while (Figure 8).

Figure 7 – View of entire result set broken down by test run

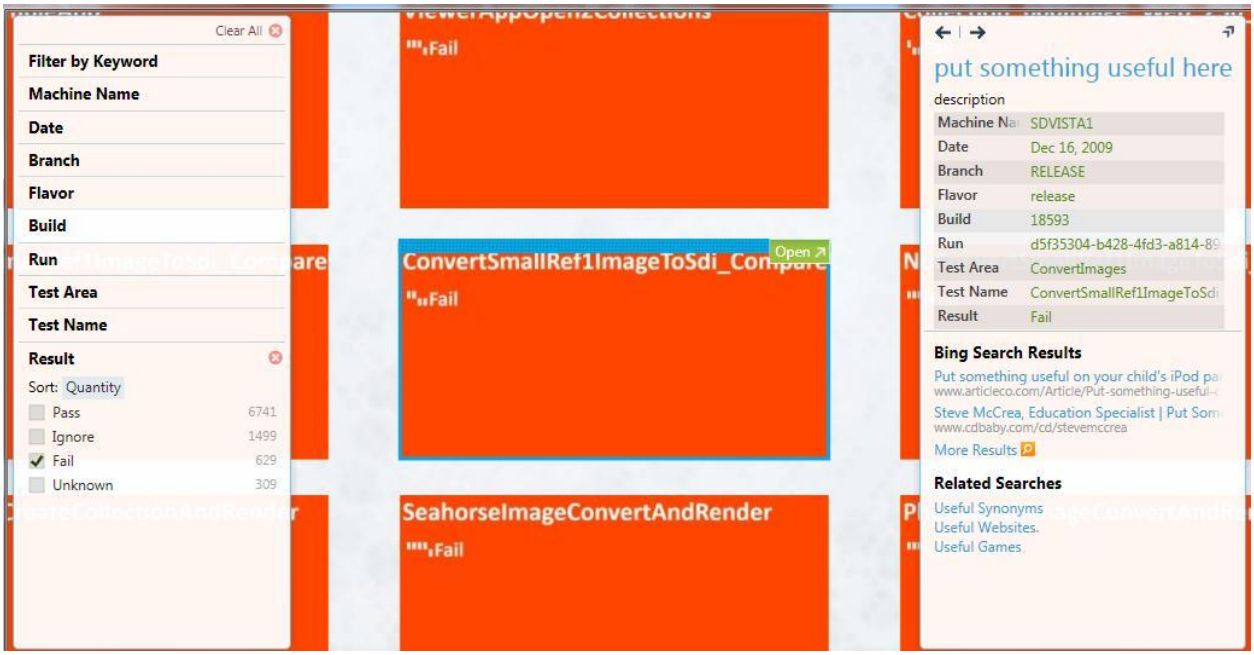


Figure 8 – Single test run result – note sparkline indicating trend of pass/fail over the last several runs. Each tic represents an individual test run for that case. The test case in the center has two passes, then two fails in a row.

4. **Code Coverage**

Code coverage analysis is often cumbersome to interpret. With the information put into PivotViewer and a simple complexity calculation, it becomes apparent very quickly where test coverage should be added.

The code coverage collection's utility came from the fact that people not used to a domain-specific tool such as Sleuth (Sleuth.exe is the executable for Microsoft's Magellan code coverage toolset) could immediately access the information. This removed the learning curve for interpreting code coverage information. Also, the ability sort and filter without authoring complex queries really made a difference in how quickly people could explore the data from various angles.

The reason we wanted a graphical representation of the data is because we wanted testers to be able pull up data quickly to determine where to focus their testing. Historical data shows that Sleuth was so cumbersome in this regard that testers we were deferring analysis and reporting to select individuals who could extract that information instead of using it themselves on a daily basis.

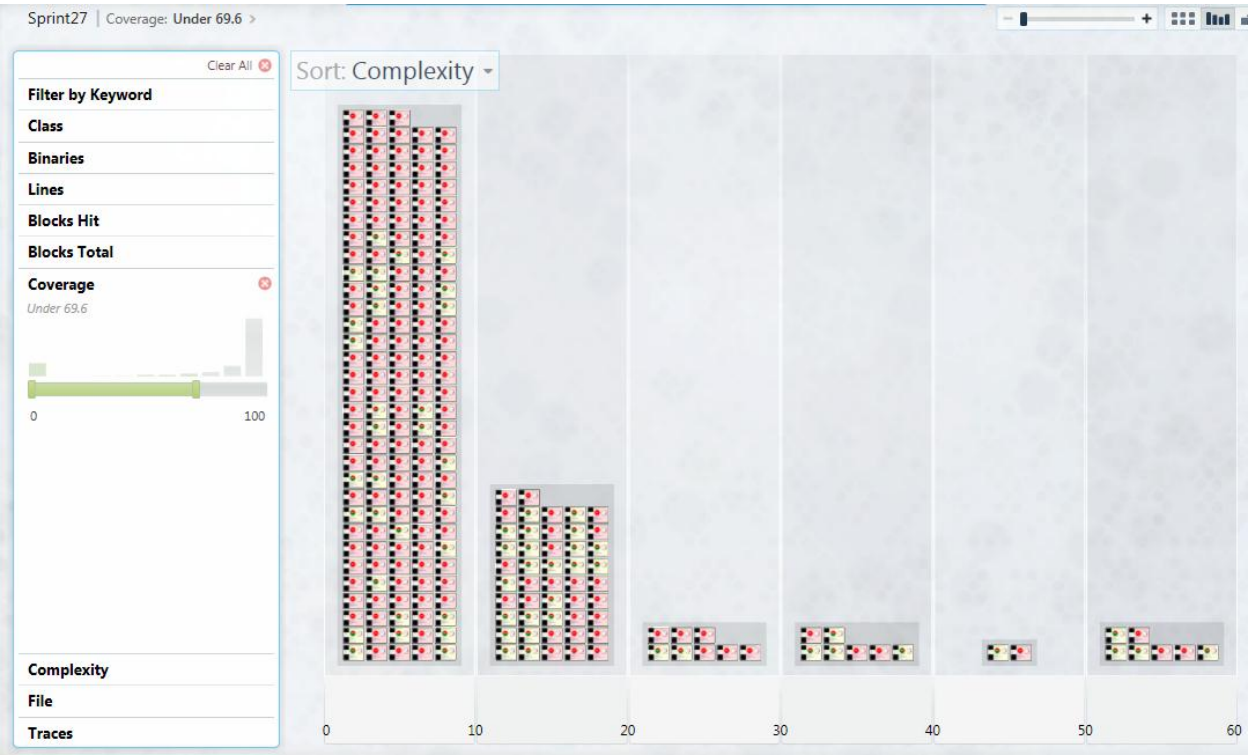


Figure 9 – Code coverage – what code should I write automation for next? The code on the right side of this histogram! Green, yellow, red (tested, partially tested, not tested) blocks distributed roughly in a ratio of 5:3:2 shows we're on track for hitting 70-80 code coverage goals. When sorted by descending complexity, we expect to see green in the high ranges and red mostly in the low ranges, indicating that risky functions are mostly tested as those would be high priority targets for testing. In this figure the view is already drilled down to the functions that have less coverage than we'd like, sorted by "Complexity".

Since the collection is color coded and uses a gauge to represent complexity, it is immediately apparent when new functions were added which were not tested (seen as clusters of red) and how much risk those represented (if the red blocks' gauge needles were high, we had a problem). See figures 9 and 10.

When sorted by file or function name, red blocks (functions not adequately tested) showing up in clusters indicates inadequate test coverage across an entire feature or component. In practice on the Seadragon project, this has actually identified new code which was not described in the check-in mail and hence was completely off the radar. This visualization raised red flags for the test team and resulted in features not relevant to the sprint being reverted from the source tree.

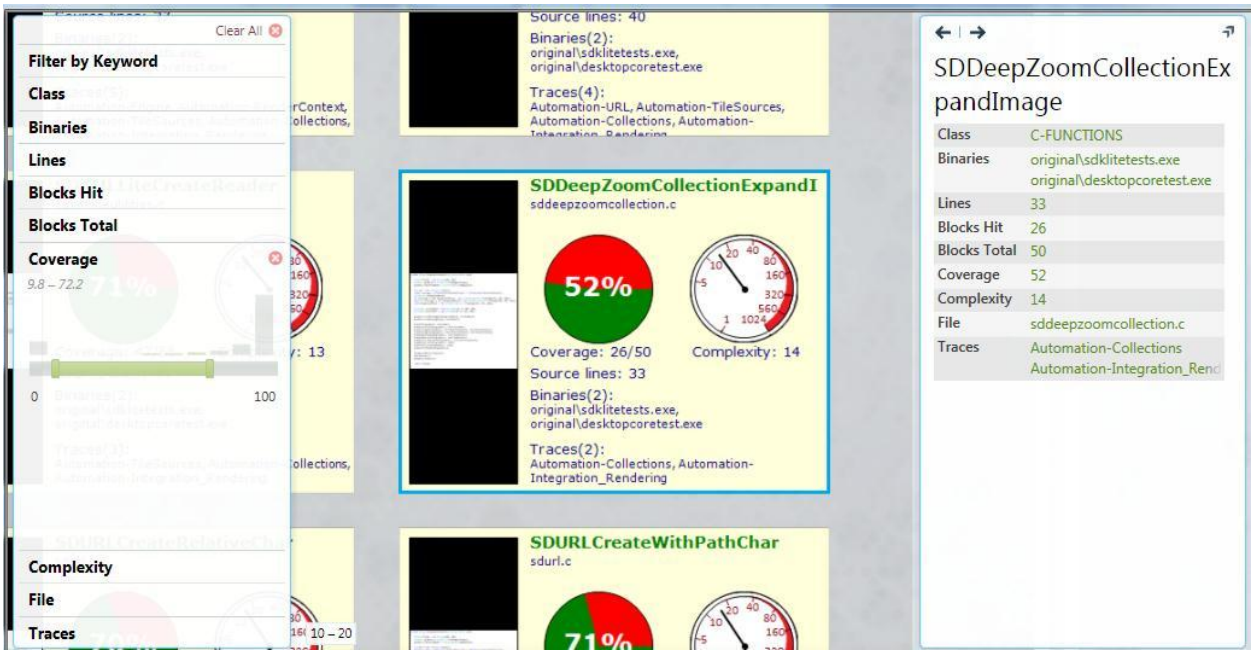


Figure 10 – Code Class with Complexity meter and code displayed

Once the PivotViewer code coverage collection was created and automated, new collections required approximately twelve minutes to be generated. Initial, one-time costs for setting up the system was approximately three days of work by one test engineer. The bug database collection had a one-time cost of five days for one tester to develop and either a few minutes or a few hours to create new collections depending on the size of the result set. The largest set of test results was 40,000 items.

In these three example collections, we've gone through three stages of learning.

1. Understanding of data quality. You can visually see where there are holes in the data. You can quickly identify what information your team needs to gather. For example, in the Bugs collection above, it was quickly obvious that my team wasn't using the Severity field consistently.
2. Visualizing trends you expect. Easily viewing any number of trends that you expect to see. For example, you may expect to see a certain shape for incoming defect reports – you can view this visually in a histogram easily and either see what you expected or be surprised by a shape you didn't expect to see. In the Bug collection, the incoming rate matched what I expected to see for the releases we had during that time.
3. Discovering trends you don't expect. With faceted browsing and searching and a visual display, it's easy to "swim around" in the data. While doing so, it's very common to discover interesting trends that you never thought to look for. An example of this was finding bugs that were resolved Fixed without

going through the proper Triage process. Another example given above was discovering code that was checked in that wasn't part of the scheduled work and therefore, wasn't yet tested.

4. How to put your own data into the PivotViewer experience

Since the following information is already covered in great detail at http://www.silverlight.net/learn/PivotViewer/, I'll include just enough here to allow the reader to understand the scope of building their own collection and refer the reader to the above URL for more detailed information. The requirements to host your data in a format that is viewable by PivotViewer are variable and depend on the type and more importantly the size of the dataset as well as whether the information needs to be real time.

4.1. Kinds of Collections

There are three primary kinds of collections. They differ in size and their ability to respond to custom user queries. Structurally, they are composed of either previously generated (or *static*) XML, or XML generated *dynamically* in response to a query. Simple collections are relatively easy to build and are recommended for collections with 3000 or fewer items and fairly static dataset that doesn't need to be updated very often. Linked collections are larger collections that are built by linking many simple collections together. It is useful for datasets that are static in nature because it's fairly complicated to update these collections. I don't recommend creating Linked collections and I'll explain just two types in the paper. Just in Time collections are recommended for large datasets that change often. A sample Just in Time server has been made available to the public. The examples in this paper represent both simple collections and Just in Time collections. The bug database collection is a Just in Time implementation. The test run results and code coverage results are simple collections because they are experiments to test the experience before investing more to incorporate them into our team's workflow. In practice, however, these collections should all be Just in Time implementations. For more detailed information on building collections, please refer to the documentation here: http://www.silverlight.net/learn/PivotViewer/collection-design/

For a Just in Time server implementation, like the Bug Database collection presented here, the focus is on creating the code that connects to your data-source and translates it to a set of items containing properties. If your data is already in tabular form, then this is straight-forward as each row maps to an item and each column maps to a facet value. However, you might want to combine multiple tables or perform aggregate calculations on multiple pieces of data.

Here are the steps to necessary:

1) Write the code to generate a list of items, each of which will have multiple facet values. If your source is a database, these might map to rows and columns in a particular table. However, you might want to join table or use a web-feed or middleware API to generate these items. Item facets might map directly to fields from your data source, or they might be calculated from aggregated data from multiple fields or tables. For example, a "Customer" item might have a "Number of sales" facet that is actually the sum of their entries in the "Orders" table in your database.
2) Create a XAML template that will visually represent your data. This should include template entries for text that you want to appear on the card, perhaps a customer's name or the title of a

bug. It can be visuals that vary based on the source data – perhaps a different color for different facet values, or an image embedded from a URL taken from a hidden facet value.

Once these two steps are performed, you should be able to deploy them to your JIT server instance and test. The JIT server requires ASP.NET and requires very little setup.

This snipped from the white paper is also pertinent: http://www.getpivot.com/developer-info/jit-tools.aspx

4.2. Collection Design

As with simple and linked collections, the PivotViewer allows you to quickly visualize the information within each collection view returned by a just-in time collection server. With this in mind, choose an appropriate number of facet categories and design your facet values so that applying them slices the view in ways that highlight interesting aspects and trends in the data. Select visual representations that are appropriate to the level of resolution displayed at any one time and which consistently support the message and insights your collection application is attempting to deliver.

With a just-in-time collection server, typically each collection view represents just a fraction of the items available in the larger backing data source. With this in mind, design your collection to include hyperlink facets that enable someone to launch a new collection view related to the current item in some way. For example, a collection view of household products containing an item for “table cloth” may provide multiple hyperlinks in the Info Pane that allow navigating to collection views associated with this item, e.g. “dining room”, “linen”, “matching items”, “blue”, “$10-$20”, or even provide actions such as “Add to cart” or “Send as email”.

With the authoring tools we have available, one can create a prototype collection that’s viewable in the Pivot client (www.getPivot.com) within an hour or so. More sophisticated collections that require a Just in Time server can take a few days to create and some backend infrastructure.

Creating a simple collection has four distinct steps:

1. **Pick your data** - First, pick a set of data to turn into a collection and decide how you want to present it. For tips, see Collection Design*. Any test data you collect is potentially interesting to put into PivotViewer. My team has created collections for Bug, Test Results, and Code Coverage. Other collections could be built with Performance, Stress, Load, or practically any data you generate while testing.
2. **Create XML and images** - Once you have your data sources, you’ll need to describe it in Collection XML (CXML) and transform your images to the Deep Zoom format. We’ve built a variety of tools* from an Excel plug-in to an open-source software library, or you could build your own. See Collection XML Schema*, Collection Image Content*, and Collection Design* for detailed information.
3. **Host it** - To share your collection with others, host it on a web server. For more information, see Collection Hosting*.
4. **Share it** - Download our PivotViewer SDK* in order to build a Silverlight control to host the collection on your webpage!

*Links in this section can be found here: http://www.silverlight.net/learn/PivotViewer/

4.3. Architecture

Conceptually, a collection is just like any other web content. There's a set of files on a server, and a local client that knows how to display them. In the current web, the files are traditionally HTML and images. In the collection case, the files are CXML and Deep Zoom-formatted (DZC) images. When the user browses the collection from a web page, the PivotViewer will use the Silverlight Control to display the files. See the following figure.

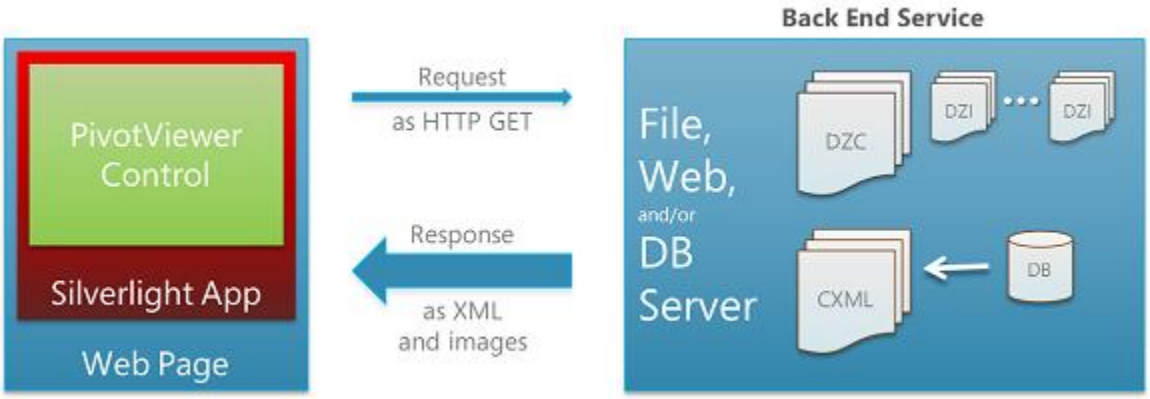


Figure 11. Basic architecture

5. Summary

I've offered a variety of ways we've thought to use PivotViewer in our testing with different sources of data as examples. Then I gave an explanation of how we built these collections and about how long it took us to create them. There are also developer documents that I've provided links for in case you need more technical information to build your solution.

PivotViewer Silverlight control and the Pivot client are available free today and you can build solutions on top of the control. I believe that the use of PivotViewer by testers throughout the industry will raise the bar for quality and give users better products.

I'd love to hear of any solutions you come up with on Getsatisfaction.com.

http://getsatisfaction.com/live_labs_pivot/products/microsoft_pivot_pivot

Streamlining test automation through white-box testing driven approach

Sushil Karwa & Sasmita Panda
McAfee, Inc. Bangalore, India
sushil.karwa@gmail.com, sasmita@gmail.com

Abstract

Software test automation has been perceived as something that can cater to our Regression or Smoke tests and if time permits, to Functional tests. How often have you found yourself in a situation where you want “something extra” that could complement your black box test automation efforts? In this paper, we propose to bring innovation to this age old approach of test automation which mostly relies on black box testing approaches. This paper does not insist on replacing black box automation, rather it proposes to complement it with white box test automation. We will talk about various white box automation ideas that can be implemented across the different stages of software development life cycle to improve the overall software quality.

This paper explores how white box test automation can fill in the gaps in your overall test automation strategy. We will also discuss how a test automation strategy that includes both white box and black box automation techniques can streamline the achievement of more effective and efficient testing. Finally, we will share our lessons learned while implementing white box test automation.

About the Authors

Sushil Karwa *is a Sr. QA Project Lead at McAfee. He has been associated with McAfee since last 7 years. At McAfee, he is responsible for managing the end to end Quality goals for the Security Management Server product. His passion towards improving product quality through white box testing saw him initiating and implementing the process of white box test automation for various product teams at McAfee. He also leads certain specific initiatives w.r.t security testing efforts at McAfee.*

Sushil has a MS in Quality Management from BITS Pilani University, India. He is a Certified Ethical Hacker and Security Analyst from EC-Council.

Sasmita Panda *is a Sr. Software QA Engineer at McAfee and works with Host Intrusion Prevention System product team. Her testing and quality assurance experience includes a focus on white box testing automation for a client-server based application. Her current role also involves identification of different techniques for measuring code coverage. In her present role, she also performs security testing with a focus to uncover security loopholes in the product at various phases of SDLC.*

Sasmita has a Master in Computer Application from Madras University, India. In past she had also worked as software developer and is very passionate about improving software quality and promoting quality achievement throughout the organization.

1. Introduction

In today's complex software environment test automation has not only become a strategic necessity but also critically vital to improve the overall product quality. The QA fraternity has started to view test automation as one of the most desired methodology for testing due to its quite evident benefits – cost saving, time saving and reliability.

Although there has to be some deterministic set of rules or guidelines on when and how to use automation, at times this is more of a decision that is influenced by the use of "common sense". If we know that a test is not going to be run repetitively, this really shouldn't be a part of the automation test suite. But what if these repetitive tests are too expensive to automate through black box automation? Do we drop them out? Or do we give it a thought that it might just be an "API call" away if it were automated using white box test approach?

In this paper we're going to talk about using various white box test automation techniques that can be used along with black box test automation.

This paper will consist of following:

- A typical test automation strategy
- Most commonly followed approach for test automation
- White box test automation and how it offers more value to your overall test automation
- Challenges faced while implementing white box test automation

2. Are we balanced?

Testing is all about trade-offs. Testing as an activity has evolved over a period of time. Today we are spoiled with choices when it comes to picking up different testing techniques. Lot of QA personnel today face this million dollar question – which testing types to use and which to avoid for your project? With so many testing types available, where we focus our time can be a non-trivial question to answer. Automation has proven its worth over several years and has answered a few of these questions. But even today, many of us find ourselves in a tricky situation to decide how much manual vs. automated testing is good enough.

Some would say we have a goal to automate n% of our manual tests. But what's the point of automating even 100% of your tests if those tests are not covering your product from all angles. We still might be unbalanced when it comes to test automation. The reason: many of us do not do any automated white box testing. Even if many of us appreciate white box testing because of the value it offers, it may still be missing from many of our testing toolboxes. We will discuss in this paper how automated white box testing can balance our overall test automation efforts.

3. Typical Test Automation Strategy

Test automation has always been a daunting and demanding task. So how do we see quick returns from test automation? Before getting into details of white box test automation, let's have a quick look of what an ideal test automation strategy looks like. There are various aspects about test automation one should balance in order to come up with an effective test automation strategy. Discussed below are few of these aspects.

3.1. Identifying your test automation goal

The first thing before starting test automation is to identify your test automation goals. Your test automation strategy would evolve depending on what you want to achieve out of test automation. Some of the typical automation goals include

- Efficiency – Regression tests, reducing test time
    - Ex: Automating a routine task like looping through various input types, automating smoke testing.
- Increased Test Coverage – Enhancing the test coverage by automating tests
    - Ex: API testing
- Multi-platform coverage – Testing the product on multiple platforms in an automated way.
    - Ex: Automating installer testing.

3.2. Test Automation Prioritization during Product life cycle

Once the test automation goal is set, we need to prioritize the tests that need to be automated. Prioritization of tests can depend on certain driving factors like:

- Repeatability – Tests that will be run repeatedly
- Complexity of test scenarios – workflows, real world scenarios, etc.
- Nature of the tests – monotonous, uninteresting, etc.
- Sometimes we prioritize new test cases over automated regression ones.

3.3 Setting realistic expectations

Test automation cannot and should not be treated as one of the main sources for finding “new” bugs. Yes, test automation will discover defects for you but not in large numbers.

Automating everything is another unrealistic expectation from test automation. We should not be looking at test automation to replace manual testing completely.

3.4. Defining measurable goals

Goals for test automation should be specific, realistic and achievable. Automating n% of the test cases for a particular feature area in your application shouldn’t be the only means of measuring the automation goals.

Automating the Build Verification Tests (BVT) test suite is something that is quite achievable and realistic. Automating the complex and mundane tests could be another specific goal.

4. Test Automation – most commonly followed approach

While the obvious benefits of test automation are widely known, test automation is usually considered to be just black box test automation. Test automation has always promised to cut costs in terms of execution time and needed resources. If applied correctly, test automation benefits can include

- Reduced cost of testing – especially when you automate the repeatable tests
- Accelerated time to market – automating tasks that are truly on critical path to improving product delivery timeline can improve software delivery time
- Reduced cost of quality – test automation support the timeline decrease of the software testing lifecycle, facilitate defect identification, and thus help improve quality

Considering the benefits offered by test automation, it has been one of the most sought after types of testing in the software development cycle. Even before we have any User Interface to test, we could still write some automated black box tests to either remotely connect to any application and execute the tests or execute the functionality from a command line provided the application supports it. During the testing cycles we come across lots of repetitive and mundane tests in addition to other tests that are prone to errors if run manually. Automating these types of tests can save much manual test execution time which can be used for other productive tasks. Building up the automated test suite not only helps increasing the confidence in the product but also enhances the test coverage.

Typically in most of the organizations, black box testing is the most commonly used approach for test automation. It's a good way to start, but it has certain limitations. Some of them are listed below:

- Possibility of redundantly testing the same functionality
- Chances of uncovering new defects are less
- Lack of internal knowledge of the code, does not help in coming up with new functional tests
- Cannot test all APIs – inability to create special hooks in the code
- Limited testing for error conditions and paths
- The time to implement automated test cases

So what is the alternative? How can we overcome these limitations and still be able to increase our overall test automation? Because white box testing deals with the internal knowledge of the code, it can offer a lot of value and streamline our overall test automation efforts. White box test automation in a real sense can fill in those gaps in your test automation strategy. Automating white box tests not only complements the black box test automation but also helps uncovering altogether different categories of defects that are hard to find through black box testing. Let's review what the different white box testing techniques are and how they can be automated.

5. White box test automation techniques- value additions

5.1. API Testing

An Application Programming Interface (API) is a collection of software functions and procedures. API tests essentially takes into consideration the individual methods and functions that make up a software system. An API test will not necessarily test the overall system as a whole, but focuses on the individual building blocks of the system and ensures that they work correctly. But generally API tests are more extensive than the unit tests. They take into consideration most likely usage scenarios that the API will be used for. This typically involves interactions between several different blocks or modules within the software application.

In our project, we started white box test automation by developing API tests. We focussed on testing all the public APIs in our code and targeted for 100% test coverage on them. We followed some general guidelines while developing our automated API tests. Some of them were:

- Public API tests should only use the classes and interfaces that are part of public API

- Ensure to write tests for all "if" and "else" statements, and "try" and "catch" blocks.

- Use assertEquals() instead of assertTrue(), since it gives better feedback over what was expected when a failure occurs.

APIs are the building blocks of your software product and improving the quality of the software largely depends on how these APIs are tested. Testing all the APIs through black box testing types is not always feasible. For instance, certain error conditions are difficult to force while testing through black box approach alone. Combining certain testing techniques like equivalence classes, boundary analysis and forced error testing with API testing proves to be very effective.

Since API testing directly targets the code level, it is one of the important types of white box testing. In order to test a typical usage scenario of an API, testers end up writing test code or mini programs. Automating API testing by writing white box tests helps you to increase the confidence in your test coverage. Here are some of the areas that need to be considered while automating API testing.

5.1.1 Test the APIs by setting up the initial conditions required to invoke them:

- Some API requires a certain set of activities to be done before it can be called.
- These initial conditions, if set before invoking the API, can influence the behavior of that API.

- Example: A non-static member function API requires an object to be created before it could be called.

5.1.2 Test an API based on its declaration:

- Create your tests based on the nature of the API.
- For example: An API can be a standalone API i.e. not a member function or it could be static / non static member function. Depending on its declaration, the object has to be created to invoke the API.

5.1.3 Test an API based on its invocation:

- An API can be called directly.
- API can be invoked by certain events, for instance mouse click, mouse movement, etc.
- API is invoked when an exception occurs.

5.1.4 Test the outcome of the execution of API:

- API when invoked may have several outcomes.
- Some returns certain data or status.
- Some just does not return anything.
- Some modify certain resources.

5.2. Code Coverage

Code Coverage is a metric that can be used to determine how much code has been executed by your test suite. It helps in identifying the paths in your code that are not getting executed. Since black box tests are designed without any knowledge of internal implementation, there would always be some part of the systems that gets rarely tested or completely missed out.

In the following code snippet, achieving 100% statement coverage through black box testing may not be easy. In order to execute the code on line number 234 the “if” condition should be true and setting the value of type to null through black box testing might be little tedious. By writing a simple white box test, this type value can be easily set to “null”. This would evaluate the condition to true and thus the code on line 234 gets executed.

```
231    public EnterceptPolicy CreatePolicy(PolicyObject po, PolicyType type, Locale locale) {
232        // if the type is null, this is a non-starter
233        if (type == null) {
234            throw new NullPointerException("type must be non-null");
235        }
```

The best way of measuring the code coverage on a regular basis is to automate the process of running the code coverage tools and generating the reports. The following steps ensures that the code coverage results get generated automatically so that one can analyze the reports and add new tests to increase the overall test coverage:

In our project we automated the code coverage process through both black box and white box testing.

- Developed scripts to instrument the build delivered to black box team using EMMA code coverage tool.

- Ran black box tests (manual and automated).

- Generated code coverage report.

- Configured EMMA with our automated white box tests in IDE.

- Executed our white box tests and generated code coverage report.

- Automatically merged both the reports using EMMA.

- Analyzed the report and came up with new test scenarios to increase coverage

- Start developing white box tests early in the lifecycle.

- Use an appropriate code coverage tool and integrate it with your white box tests.

- Run the automated white box test suite on a regular basis (probably on every build)

- Automatically generate the code coverage report after the execution of the automated white box test suite. By automating this process you make sure that code coverage report always gets generated which can be analyzed later. Either write some batch scripts or add new targets in your build file to automatically generate the code coverage report upon execution of white box tests.

- Analyze the coverage report and create new test scenarios. Develop additional white box and black box tests for these test scenarios

- Based on your coverage goal, strive for adding new tests to cover the untested code and meet the coverage objective.

5.3 Code Analysis

Code analysis is an automated way of analyzing the components and resources of the application under test to identify situations that are likely to produce errors. Code analysis tools scan the code and look for potential problems based on some pre-defined rule sets. For instance the “unused code” rule set would contain a collection of rules that find the unused code in your application. Similarly the other most common rule sets would report errors like buffer overrun, uninitialized memory, null pointer dereferences and memory and resource leaks.

Some of the most prominent benefits of Code Analysis are:

- Finding probable bugs in the code.
- Locating the “dead” code.
- Detecting performance issues.
- Improving code structure and maintainability.
- Helps in conforming to coding guidelines and standards.
- Automated tools provide mitigation recommendations, reducing the research time.
- It allows a quicker turn around for fixes.

There are tools available in the market that can be integrated with your development environment. One can create automated targets for running code analysis tools on their product code and include them in the build file. So whenever the code is built, the code analysis target will also get automatically executed and the report gets generated. Some IDEs like Visual Studio come with an option to automatically perform the code analysis and generate the report every time the code is built.

After the report is generated the findings need to be analyzed manually and defects should be raised appropriately.

In our project we used a code analysis tool called PMD whose plugin is available for several IDEs.

- PMD supports creating targets in Java based build tool like Apache Ant.

- We created ANT targets to automatically run code analysis on individual modules of our code

- These ANT targets used to get executed every time we ran our white box tests.

- Code analysis reports were generated for every run automatically.

- These reports were then manually examined and defects were raised appropriately.

- We also automated the process of detecting any API violations in our code.

In addition to the pre-defined rule sets provided by the code analysis tools, one may want to add their custom rules to fulfil specific need. For instance public API tests should only use the classes and interfaces that are part of the public API with exceptions of some third party libraries that are used in the project. If this is not the case you should treat it as a case of API violation in your code. Creating a customized rule set to detect such problem and feeding it into the code analysis tool would automatically discover this problem in the source code.

5.4 Build Verification Testing

Build Verification testing (BVT, also known as smoke testing) is used to make a decision whether a build released to QA is in an acceptable state so that detailed functional testing can be carried out on that build. The BVT test suite is comprised of test cases that test the basic functionality of the product. If any of the tests fails from the test suite, the BVT fails. Minimizing human intervention in test execution is the key here.

BVT is executed every time a build is released to QA. Considering that it needs to be executed repetitively during the product life cycle, it is one of the best candidates for test automation. BVT can be a blend of both black box and white box tests. Black box tests can focus on the UI (User Interface) part and make sure that the UI conforms to the product usability specifications. White box tests can test the functionality by directly calling the relevant APIs.

Automating BVT requires setting up a unit test framework that can be used to make direct calls to the code. One can create a BVT level white box test suite that gets executed automatically whenever a build is released to QA. Based on the result of BVT white box test execution, the build can be straightaway rejected without any manual intervention required, thus saving lot of testing time during the project.

5.5 Security Testing

Security testing uncovers any security vulnerabilities in the application that can be exploited by malicious user. The main objective of security testing is to ensure that the application under tests remains robust even in the face of a malicious attack.

The functional specification might outline a secure design, the developers might be attentive and write secure code, but it’s the testing process that determines whether the software is secure in the real world. Through white box testing one can validate the implemented security functionality and uncover the vulnerabilities that can be exploited.

Security testing can be very time consuming and may sometime go beyond the scope of overall testing cycle. One should follow a risk-based approach while performing security testing. Risks, once identified, should be ranked based on its severity, business impact and probability of occurrence and

prioritized accordingly. Testing efforts should be made to focus on high risk areas. In order to have optimum coverage within stipulated time, one should use both black box and white box testing techniques to carry out security testing on their product.

In our project we automated security testing by developing framework using Ruby and WATIR.

We created automated tests to pass in malicious scripts as inputs to our application.

We tested for vulnerabilities like Cross Site Scripting, SQL Injection, URL Manipulation, Information disclosure, etc.

We also developed automated tests to verify the implementation of security functionality in our product like encryption and data storage.

Around 15% of our security defects were found by our automated white box tests.

These automated white box tests were used for regression testing as well.

Automating security tests through white box not only saves time but also provides new avenues for performing security testing in other unexplored areas. One of the examples of how security tests can be automated is discussed below. Let's say your application interacts with the database and we know that improper input data validation can lead to SQL injection attacks. A white box test could be to access the object that holds the input data and provide a maliciously crafted SQL string to delete a table in the database. A security test would be to ensure that the table does not get deleted when this data is stored in the database. Similarly one can think of automating security tests to uncover coding or functionality errors to name a few as mentioned below.

- Improper input validation compromising security.
- Broken authentication
- Insecure cryptographic storage.
- Information disclosure vulnerability
- URL Tampering
- File manipulation
- Manipulating Sessions

6. Challenges

There are certain challenges one may face while implementing white box test automation. Some of these are discussed below.

Skill set requirement: In order to perform white box testing, a specific skill set is required which include basic understanding of a programming language. Other skills include familiarity in designing framework, working with tools, understanding various white box testing techniques, etc.

Expecting too much in the beginning: Understanding the code, developing the basic white box test automation framework, getting familiar with the tools used, ability to find defects in the vast sea of code are some of the reasons why it takes time to see any visible results at the beginning.

Not defining the scope of the automation: Without defining the scope of white box test automation upfront, one cannot achieve the desired results. Automating 100% should not be the goal. By automating we are not going to eliminate the manual testing completely.

Dependency on development team: There will be some kind of dependency on the development team to understand the workflow of the code and the way it's been implemented. Sometimes the design document may not provide you the information you are looking for.

Maintaining the energy: Maintaining the motivation and enthusiasm to do white box testing can be a big challenge. This should be carried out with self-interest rather than being forced by the management.

7. Lessons learned

We feel like we learned the following lessons from our experience of implementing white box test automation in our project.

- It's very important to use right tool and technology that suites your need while performing any of the white box testing techniques.
- Interaction with developers helped us to come up with good ideas for our white box test automation.
- Starting white box test automation from the early stages of the product development cycle enabled us to uncover defects in the code that could be fixed in cost effective way.
- Keep searching for new ways to improve the process and technologies you use to make things more efficient, productive and simpler.

8. Conclusion

We found white box testing can complement black box testing to increase overall test effectiveness. White box test automation can be a very powerful mechanism to test your application. It helps in uncovering programming and implementation errors very early in the life cycle. Employing the right white box testing techniques to effectively test your application can help increasing your confidence in your product significantly. By complementing the two forms of testing, more areas can be covered and new tests can be developed to increase the overall test coverage. As the experts say, the earlier a defect is found, cheaper it is to fix it. This is exactly where white box testing plays a big role to save money and automating it saves the testing time.

References

[1] Henk Coetzee, Volume 7, Number 2, March/April 2006, "Software Test Automation in the SDLC" - http://www.testfocus.co.za/

[2] Jussi Kasurinen, Ossi Taipale, and Kari Smolander, Volume 2010 (2010), Article ID 620836, "Software Test Automation in Practice: Empirical Observations" - http://www.hindawi.com/journals/ase/2010/620836.html

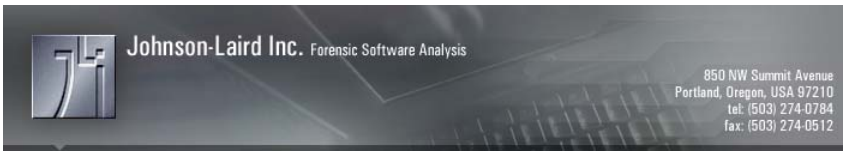
[3] James McCaffrey, "API Test Automation in .NET" - http://msdn.microsoft.com/en-us/magazine/cc163892.aspx

[4] Automated Software Testing Magazine, May 2009 Edition - www.automatedtestinginstitute.com

[5] List of rule sets used by code analysis tool PMD - http://pmd.sourceforge.net/rules/index.html

QUALITY PEDIGREE PROGRAMS: OR HOW TO MITIGATE RISKS AND COVER YOUR ASSETS

By
Barbara Frederiksen-Cross (barb@jli.com),
Marc Visnick (marc@jli.com), and
Susan Courtney (susan@jli.com)



ABSTRACT

Forensic software analysts routinely review source code in the context of litigation or internal software audits to assess whether, and, to what degree, a body of software uses or references third-party materials. These references may include source code examples incorporated directly into a program, source code routines that are statically linked as part of the program, the use of binary libraries that are dynamically referenced when a program is executed, or URL-based citations to third-party materials, such as an article on a website. While third-party materials are obviously invaluable to software development, third-party materials may introduce a variety of legal or security risks into software and expose a company to unexpected legal liability and/or negative publicity. Thus, quality software is defined not just by technical measurements, but also by the presence of a comprehensive set of policies and procedures that help mitigate these potential risks.

We believe it is essential that companies proactively establish a baseline pedigree for their existing software via a *forensic code audit*. This act authenticates a moment in time where all known third-party materials are appropriately catalogued, and risks associated with those materials are fully understood by the company's relevant business and legal stakeholders. But a one-time pedigree analysis is not sufficient. A pedigree analysis should be part of a comprehensive, *quality pedigree program,* a set of well-defined prophylactic policies and procedures surrounding the use of third-party materials. These policies and procedures should take into account the entire software lifecycle, including any customer support obligations that may remain once a program is deprecated. A company that proactively implements a quality pedigree program is better positioned to respond to customer requests, react to lawsuits or potential licensing problems, or to justify a particular valuation of their intellectual property in the context of a merger or acquisition.

BIOGRAPHY

Barbara Frederiksen-Cross is the Senior Managing Consultant for Johnson-Laird, Inc., in Portland, Oregon.

Marc Visnick is a forensic software analyst and attorney based in Portland, Oregon, and a senior consultant with Johnson-Laird, Inc.

Susan Courtney is a forensic software analyst and has worked as a consultant for Johnson-Laird, Inc.

Published at the Pacific Northwest Software Quality Conference 2010

1 INTRODUCTION

Modern source code development is rarely devoid of third-party content. The reason is simple (and is guided by the same principles that govern modern manufacturing): it doesn't make sense to reinvent the wheel for every automobile you build. Programmers are keenly attuned to this reality and routinely re-use existing source code when developing new products. Cutting and pasting code from past development projects, Open Source repositories, third-party tools, or code found on the Internet accelerates new development. The use of such reusable solutions saves time reduces debugging effort, and helps build more reliable code.

In addition to reusable solutions, a company that accepts such projects (make no mistake about it, that would be nearly all companies) inherits new responsibilities. The failure to effectively manage the use of third-party code can result in entanglements that open the door to litigation, trade secret loss, security vulnerabilities, and costly remediation efforts. The use of third-party source code becomes problematic when software developers write software that references third-party materials and fail to document such usage, fail to understand or comply with licensing restrictions, fail to consider copyright, trade secret, or patent entanglements, or fail to consider security implications.

Each of the problems listed above can occur whether the third-party code is proprietary code or Open Source code. We are all familiar with legal problems that arise when employees re-use code from past employers or consultants re-use source code that was initially written for a competitor. Disputes over copyrights or trade secrets can arise even with code developed under commercial licenses. The potential for problems expands exponentially for Open Source, due in part to the fact that it is readily available to developers without any authorization from (or visibility to) management, and in part due to the proliferation of different licensing models that may apply to Open Source. These licensing models may introduce significant legal complexities for commercial organizations, as they may contain terms that render them incompatible with commercial use or which contain terms that require making available any source code derived from code under an Open Source license.[1]

Use of third-party code (Open Source, public domain, licensed, or purchased) may also foster technical exposures such as security vulnerabilities. For example, if the security of an Open Source component is compromised, an individual who is aware of its security vulnerability may exploit the vulnerability to gain unauthorized access to the program function or to the computer system upon which it is running. Since the same open source routine may be incorporated into many different products, would-be intruders may find a wide range of targets that all stem from the same vulnerability.

Even if the third-party code is free of security vulnerabilities, its use may still incur security risks due to license terms that require public availability for the source code of its derivative product. Publication of source code gives would-be intruders an excellent opportunity to search out security vulnerabilities that may exist in the product. It also gives competitors the opportunity to study all aspects of the program's operation in intimate detail, including any trade secrets the program may embody.

Even if license terms do not require making source code available, the security ramifications of other license requirements, such as attribution, should be carefully examined. The knowledge that particular software includes as constituent parts specific Open Source or third-party components necessarily reveals some aspects of its internal operation, with the result that the particular software is rendered more vulnerable to tampering or reverse engineering.

2 MANAGING THE USE OF THIRD-PARTY CODE

The problems that can stem from third-party code entanglements get worse over time. As comingled code evolves, the fact of the comingling is easy to document, but the actual origin of any particular code

[1] For more information about the various types of Open Source licensees *see* http://www.gnu.org/licenses/license-list.html

component becomes harder to ascertain. Since modified code may be re-used and spread to other products, it becomes harder to identify the scope of the original code's use, and harder to contain or correct any damage or risk. The costs associated with tracking the true origins of third-party code will also increase over time, since the evidence of the code's origin may be further eroded by subsequent modifications. Further, the original third-party product may be superseded by subsequent releases, rendering code matching exercises more difficult.

Management of Open Source and third-party content is best accomplished through prophylactic procedures, not reactive panic. Companies should proactively establish a *quality pedigree program* that introduces a series of "best practices" which include:

1) carefully considered guidelines for using third-party materials,

2) a clearly defined review and approval process,

3) thorough documentation of third-party code use,

4) the periodic performance of code audits to detect the presence of third-party material in a body of source code.

2.1 THE CORE TEAM

The skills required to effectively manage third-party source code dictate the need for a core compliance team that includes participants from both legal and technical backgrounds, including input from individuals responsible for business development strategy. Management's first step to bring third-party code usage under continuous control is to charter this team.

The compliance team should be led by a single compliance officer who assumes overall responsibility and final authority for the review and approval for use of third-party code (including Open Source). Business development managers and other stakeholders help the team prioritize products for analysis, and identify critical timeframes relative to new product rollouts or system implementation dates.

One or more participants with a legal background review license terms and assess legal risk factors relating to compliance and remediation efforts. Legal team members may also assess the effect of third-party code use on internal and external contracts, user licenses, and intellectual property matters relating to trade secrets, copyrights, and patentability.

One or more team members with a technical background review candidate third-party code, and assist in determining the scope and nature of existing third-party code usage. Technical team members also help assess risks related to Open Source security and whether its proposed use is appropriate; *i.e.* determine if the terms of license restrictions for commercial use or restrictions requires the publication of code that embodies trade secrets or other proprietary technology.

2.2 PROACTIVE CODE MANAGEMENT

Proactive management of Open Source and third-party content should include a published policy that defines where and how Open Source and third-party content may be used. This policy defines the criteria that distinguish between approved and unapproved licenses and code origins. The management policy should have a clearly identify scope with respect to software developed by in-house developers, independent contractors, software vendors, and development partners. Where necessary, the management team should update contract language to addresses code developed by business partners, contractors, consultants, and outside developers.

Software pedigree documentation, which identifies the origin and use of all third-party components, should be formalized and produced as an artifact of all new development projects. These records should be maintained and updated throughout the lifecycle of the software to account for any changes to third-party content. Think of these records as insurance or as a metaphorical *get-out-of-jail-free* card that guards against expensive litigation.

The policies governing third-party code management should include an approval process for the evaluation of third-party material, its associated documentation identifies the code origin and license restrictions on its use. The policy governing the review should clearly document the review submission process, including the expected turnaround for approval, and the communication mechanism that will be used to notify the applicant about the result of their request.

Approval from the review process will optimally be obtained prior to implementing new third-party code. It is crucial to define a review process that is sensitive to your company's business needs, development model, and risk tolerance. The review process itself will typically include identification and assessment of the proposed usage, applicable license terms, legal considerations relating to licensing and Intellectual Property ("IP") ownership, and a review of any known security considerations with respect to the software or its use.

To facilitate the approval review, engineering and software development teams should provide advanced notice to the review team about the proposed third-party code use. The notice should identify the software, its origin, the proposed scope of its use (*e.g.,* experimental, internal use only, development tool, distributed with product, etc.). To expedite review, a copy of the actual license text is attached to the document for the review process. The review document and the license should be preserved for future reference.

Although this process may sound cumbersome, streamlining is easily achieved by the use of black lists, white lists, and grey lists. These lists will grow over time and represent classifications of licenses and source origins into one of three categories:

- The *black list* identifies materials that must never be used (e.g. materials developed for previous employers, materials whose source cannot be determined, and materials subject to unacceptable license restrictions);
- The *white list* identifies pre-approved materials (e.g. license terms and origin have already been accepted, and scope of proposed new use is unlimited or precisely defined);
- The *grey list* identifies materials that are candidates for use, but require approval to assess license terms in the context of a specific proposed use.

The approval review process includes maintaining a special repository for approved third-party code and associated licenses as well as documentation to preserve a clear record of both the original third-party code base and the license under which it was obtained.

Once code has been approved from the white or grey lists, it is also important to maintain records of where, in the larger context of a company's software, that particular code is actually used, because the scope of use may grow over time. The usage records should provide information at the program or product level that identifies the specific version of third-party code used, when it was implemented, the scope or context of its use, *e.g.* internal vs. external, etc. Additionally, the records should document which specific components incorporate the code and/or which components interact with it. Further, if at all, possible individual source code files should incorporate a comment block that documents any restrictions that might apply to use of the code within a file.

Taken together, the repository of approved code and the records related to its use form the basis for technical due diligence reporting that may be required in the context of mergers, acquisitions, and divestitures. These records will also be extremely helpful in the event of downstream problems that might arise with respect to reliability, security vulnerabilities, licensing issues, patent disputes, or other legal issues related to intellectual property or product liability.

Once third-party code management policies are in place, employees, contractors, and consultants must be educated about their obligation to protect company assets by using said policies. This includes training all developers (and their managers) in the actual mechanics of the review and approval process.

One point cannot be overstated: developers must be explicitly instructed to never reference or incorporate any code from their private libraries that was developed while working for past employers. Consider adding this policy to your company's internal code review process, and provide periodic reminders about this policy to employees and contractors involved in software development.

In the authors' experience, it is extremely helpful to implement some form of regular, periodic audits or other monitoring process to identify potential problems that may arise during early implementation of the new policies. The policy should also include exception procedures to address existing code or other special circumstances.

3 LEGACY CODE

The development of third-party code management policies is directed toward preventing future problems, but what about legacy code? Open Source and third-party code bases have been available to developers for many years. Despite this, only 22% of companies surveyed reported their organizations have explicit management policies/procedures in place.[2]

The results of this survey suggest that many companies should begin their source code management program with an audit of their existing code base. The code audit, in combination with carefully considered guidelines for using third-party materials, and thorough documentation of such use, completes the foundation of a proactive source code management program. The combined use of these three practices establishes a permanent record ("pedigree") documenting the origin of your company's software.

The existence of software pedigrees helps to manage the risk of unforeseen legal entanglements. A company using a third-party code management process is better positioned to react quickly in the event of a lawsuit, a licensing problem, or to support efficient due diligence in the context of a merger, acquisition, or divestiture scenario. A proactive management plan for third-party source code can help protect intellectual property, and it can also help identify software that includes components known to have security vulnerabilities.

Since software may be modified in response to changing business needs, any proactive plan must include processes that apply to ongoing development and maintenance, as well as the initial code development.

3.1 CODE AUDIT

As noted above, the first step toward managing third-party code use is often an audit of the existing code base. The code audit has two primary goals: first, to determine whether third-party material is present in a code base (a technical exercise); and second, to determine the ultimate origin and license associated with those third-party materials (both a technical and a legal question). The audit seeks to answer very basic questions: What third-party code are you using? Where did the third-party code come from? What license(s) pertain to its use?

[2] Based on a survey conducted by Black Duck Software, Inc. at the SD West conference (March 11, 2009). SD West is an international technical conference for software developers. Black Duck Software, Inc. provides services and software to help companies manage their use of Open Source and Internet-downloadable source code.

The code audit itself does not remediate problem source code. Rather, it serves as an effective means to detect potential problems, providing useful information about the nature and scope of third-party code entanglements, and identifying applicable licenses.

One of the first orders of business in performing a code audit is to determine the scope of the audit. Scope decisions will define both the breadth (which applications) and the depth (which types of analysis) are appropriate to the business needs. It is important to note that no software audit can provide a 100% guarantee that *all* possible third-party issues will be discovered in a code base. This is especially true regarding code that is derivative of, but nonetheless textually dissimilar to, third-party code. A simple audit may comprise only a surface scan of a code base, looking for obvious indicators of third-party usage, *e.g.* to discover whether there are third-party copyright or license notices in the code. Deeper audits may incorporate more sophisticated tools to identify subtle evidence of copying. The scope of the audit is determined on a case-by-case basis, and is consistent with the circumstances triggering the audit, available time, risk tolerance, and cost factors.

Typically, the code audit uses a combination of automated tools and visual inspection to determine if third-party materials have been incorporated into proprietary code. Depending on circumstances, a variety of tools and techniques may be used. The software tools used for auditing range from simple search tools used to identify indicia of third-party origin, to sophisticated tools that compare one body of code against another. Cases that involve non-literal or structural analysis may require the use of specialized tools and techniques that render visible the architectural structure and program control flow.

One very common technique used for software audits is textual searching. Textual search strategies can reveal indicia of third-party origin, such as copyright or license references and annotations, that identify code which was “adapted from”, based on”, or “stolen from”[3] some third-party source. Although extremely useful, the results from textual searching depend on the premise that there are relevant comments embedded in the source code, and that the list of terms used for searching is sufficiently broad to identify all third-party code. For this reason, textual searching may not be sufficient for a thorough audit, especially if the body of existing code is very large. It can, however, be extremely valuable to help form a preliminary assessment about the number and nature of third-party products entangled in the source code.

The results of textual searching are often refined and supplemented by using code matching tools which serve to identify literally-similar or near-literally similar code use, *i.e.* cases where third-party code has been incorporated with no or little textual modification into code. Some code matching tools can perform an automated comparison of entire code bodies against extensive repositories that contain source code from many different publicly available sources.[4] Because such tools actually compare code, rather than merely searching textual content, they can catch third-party code use that might otherwise remain undetected.

Even with the use of very sophisticated tools, human analysis is still required to weed out false positives, catch false negatives, and, above all, to judge the contextual relevance of a hit on third-party materials. Human judgment is also required in assessing the degree to which third-party code may present a risk to security, patentability, trade secrecy, or control of intellectual property.

---

[3] Most programming languages allow programmers to annotate the source code with non-functional comments. Such comments often contain information about the purpose, origin, and operation of the code. In the author’s experience, it is not uncommon to see the phrase “stolen from” in comments that reference the origins of coding routines adapted from third-party sources.

[4] E.g. Palamida™ (http://www.palamida.com/), OSRM (http://www.osriskmanagement.com/), and Black Duck™ (http://www.blackducksoftware.com/). All these companies have tools that compare a body of source code against a repository of code fingerprints generated from a variety of publicly available software projects. It is important to note that these tools will not catch contamination from non-public sources (unless such materials are expressly incorporated into a custom fingerprint repository per customer request). Because they do not fully analyze context, these tools may also generate false positives. In the author’s experience, these tools must be accompanied by human analysis.

A thorough search for third-party materials should also extend beyond source code libraries to include executables, documentation, and other types of files. In the authors' experience, third-party materials are found not just in the obvious source files or binary redistributables, but also in a variety of potentially unexpected locations. For example:

- Container files such as Zip, RAR, Java JAR, MSI and CAB files: These types of container files store material in an internal format that cannot be inspected for third-party materials directly without first performing an expansion or extraction of the files' contents using an appropriate tool. The authors have encountered numerous instances where a seemingly innocuous container file turned out to contain unexpected third-party subcomponents, including third-party source code files under copyleft[5] or proprietary licenses.
- Archives within archives: In some instances, unexpected third-party sub-components may be buried in "archives within archives." It is essential that, before analyzing any materials set, multiple recursive passes are made through the materials to decompress all archives embodied within the materials.
- Font files: Many font files are, in fact, distributed under copyleft licenses.
- Databases and stored queries: We have found various instances of third-party routines embedded into otherwise innocuous SQL files, or saved in databases as stored procedures and queries.
- Program documentation: third-party material such as code samples may be found in program documentation.
- Generated code: Files generated by a third-party tool, such as the GNU bison-generated parser files[6] may be subject to unexpected license terms.
- Referential citations to third-party materials under a potentially problematic license: We have observed a number of instances wherein a developer includes a comment in a source file citing to a third-party location as "the inspiration for" or the basis of the source code related to that comment.

This list is by no means exhaustive. The critical point is that a comprehensive code audit must take the totality of the materials into account.

Regardless of the tools used, the product of a code audit should be a detailed record documenting every instance of third-party code, where it was found, what purpose it serves, and whether the code is used internally or distributed as a part of some product. These records should be sufficiently detailed and identify every file that includes third-party material. Where possible, the records should also preserve any information gleaned about the code's origin, owner, copyright, license terms, and the specific third-party product or project from which the code was derived. This file-level metadata will prove invaluable in the event of litigation.

3.2 LICENSE REVIEW

Once third-party code inclusions have been identified, the next step is to identify the code origin and the applicable license conditions. In many cases, comments within the source code itself will identify the specific governing license or the software product and owner. Armed with this information, licenses may be located via simple Internet searches, and, with luck, their complete text may be downloadable. In other instances, the search for source code origins and the associated license text may present unexpected

[5] Copyleft is a general term for license restrictions that require that users who use, modify, or extend a particular Open Source program must make available the source code of the modified work.

[6] Bison is a parser generator that takes a grammar description, and converts that description into a C-language program to parse that grammar. See http://www.gnu.org/software/bison/ [visited 6/25/2009]. Bison-generated parser files are expressly distributed under the terms of the GNU General Public License ("GPL") but with an exception that states: "As a special exception, you may create a larger work that contains part or all of the Bison parser skeleton and distribute that work under terms of your choice, so long as that work isn't itself a parser generator using the skeleton or a modified version thereof as a parser skeleton."

challenges. Sometimes the source code may have an obvious third-party attribution, but may come from a company that does not appear to have posted any of its software or documentation on the Internet. This is not necessarily a contradiction, as it may reveal that an outsourced developer is recycling code written for prior clients. In other cases, the software may reference companies (or individuals) whose assets have transferred as a result of mergers and acquisitions, or products that are no longer publicly available.

Your own company's acquisition of source code assets may also prove problematic, if the software pedigrees lack details at the program-specific level. This is especially true if the original copyrights were not supplemented or if the acquired code includes unexpected third-party content.

It is common for Open Source code to be expressly distributed under a choice of licenses, and found during the audit in a context that offers no guidance as to what license applies. In other cases, multiple licenses may apply due to the collaborative nature of the code's development.

The product of the license review should be a detailed record that includes the full text of each license, and where and when it was obtained. As described in greater detail below, it will be important to maintain some enduring record that establishes which license(s) correlate to which programs or code portions. This reference is essential because the compliance evaluation must be done on a program-by-program basis, since software licenses may specify different terms for commercial and non-commercial usage.

3.3 Determining Compliance Status

Once third-party code and any associated license(s) have been identified, assessment of compliance status and identification of related technical issues may proceed. This process has two primary components: a legal assessment of the current compliance status and a technical assessment of factors related to the code and its use.

The focus of the legal assessment is a review of license terms governing each third-party source code use to determine its compliance requirements, and a review of current practices to determine the current status with respect to those requirements. In addition to reviewing accounting records for third-party software licenses, a compliance assessment will likely require review and revision of the following:

1) end user license agreements or product packaging (to assess attribution compliance);

2) product revision and software download practices (to determine if required notices are shipped with each version of the product); and

3) web or phone based download support (to review the process whereby GPL-based source code can be requested or delivered.)

The technical assessment provides program-specific information that must be considered during the legal assessment. For example, the same license may contain separate compliance requirements for commercial and non-commercial usage, thereby necessitating a technical evaluation to determine the context in which the third-party code is used. Additionally, the technical assessment should address whether code based on third-party software would pose security risks, or compromise trade secrets, if license compliance required publication of the source code.

Original (unmodified) copies of any Open Source or third-party code should be preserved. If remediation requires the removal of the third-party code, a comparison of the existing code to the unmodified third-party base can expedite development of a roadmap that identifies the components which must be replaced. Conversely, if the third-party code remains a permanent part of your product, the original base code may be extremely helpful in the context of patent, trade secret, or copyright disputes. In the case of Open Source, the complete license text, as well as, any documentation relating to origin and use should be maintained in the same repository used for the base code. This practice can also be followed for proprietary third-party source, assuming that source code is available under the terms of the license agreement.

Careful record keeping during the review process is essential, and will provide a foundation that minimizes the burden of future reviews. At a minimum, the record keeping should identify where compliance has already been achieved and the status of any potential compliance issues that require further research or action. Depending on the depth of the review, the review records may also be used to flag issues relating to intellectual property protection, Sarbanes Oxley[7] compliance, or similar categories of concern.

Generally speaking, the compliance recordkeeping is program-centric or product-centric rather than license centric, since the same piece of third-party code may be re-used multiple times in different contexts. For example, third-party code may be both used internally and as part of a distributed product, and is, potentially, subject to different license terms, depending on the context of its use.

3.4 REMEDIATION

The specific steps taken to address compliance related to third-party code use, *i.e.* remediation, must consider both the needs of the business and the terms of the license. Where remediation is necessary, it will generally fall into one of the following categories:

1) Modify existing documents and processes to comply with the license terms;

2) Remove or replace the third-party code;

3) Try to negotiate for more acceptable terms.[8]

As with any business decision, factors of cost and risk will influence the decision of whether and how to remediate the issues that arise from third-party code use. Legal factors, such as license terms that prohibit all commercial use, may drive the decision process. In other cases, additional technical or legal fact-finding may be required to support the decision making process. Technical factors, such as the importance of the third-party code, the cost to replace it, the degree to which the code has been modified for integration, and resource or time constraints, may limit available remediation strategies.

Unfortunately, there is no general rule to guide retrospective remediation– every license and code use will need to be addressed on a case-by-case basis. As each case is decided, a record of the specific actions taken for remediation, as well as the factors considered, should be preserved.

4 EXPERIENCES

Distilled from several hundred forensic code audits performed in response to litigation matters, technical due diligence for mergers and acquisitions, and internal code audits to proactively identify third-party code use, the following observations are offered:

- Nearly every forensic code audit performed by the authors of this paper has revealed evidence of undocumented third-party material in the subject code base.
- Automated analysis tools, while useful, are not substitutes for human analysis; the human brain is ultimately the best judge of contextual relevance.
- A company's disclosures regarding third-party materials almost always prove incomplete. This suggests that many, if not most, companies have yet to both understand fully this vulnerability and to implement effective pedigree tracking practices. While representations and warranties are certainly useful, our guiding axiom is "trust but verify."

[7] The legislation came into force in 2002 and introduced major changes to the regulation of financial practice and corporate governance. For more information *see* http://www.soxlaw.com/index.htm

[8] This form of remediation may be successful if the negotiation occurs before the code is actually used. Its likelihood of success diminishes once the third-party code is incorporated into live software.

- Education is crucial; the authors have seen multi-million dollar litigation launched in response to the actions of a single uninformed developer.
- A proactive audit can help identify potential problems. In acquisition scenarios, the authors have seen deals terminated because of the unexpected scope of third-party materials in a seller's code base, usually in contrast to the seller's disclosures.
- Proactive audits can also have big payoffs in the context of mergers and acquisitions. The enduring value of your IP assets depends on proper identification, disclosure and management of third-party content; moreover, sophisticated buyers may force last minute discounts when the seller can not comply with technical due diligence disclosures.
- The validity and enforceability of copyright registrations depend upon appropriate identification of pre-existing works, including Open Source and third-party components.

5 CONCLUSION

Successful, profitable, and cost-effective management of Open Source and third-party content is best accomplished through prophylactic procedures, not reactive panic. Almost all of the pedigree problems of software can be managed, and the pitfalls described in this paper avoided, with proactive care. "Best practices" include 1) carefully considered guidelines for using third-party materials, 2) a clearly defined review and approval process, 3) thorough documentation of third part code use, and 4) the periodic performance of a code audit to detect the presence of third-party material in a body of source code. The use of periodic audits serves to demonstrate compliance with usage and documentation guidelines, or to identify any third-party usage which falls outside such guidelines. The combined use of these four practices creates a quality software pedigree that helps to manage the potential risks associated with use of third-party code and may avert unforeseen legal entanglements. A company using this type of program is better positioned to act quickly in the event of a lawsuit, a licensing problem, or to support efficient due diligence in the context of a merger or acquisition.

6 APPENDIX ONE – CODE MANAGEMENT CHECK LIST

This checklist is designed to help foster dialog and awareness between legal and technical participants in the code management process. While not exhaustive, this list may be helpful to identify opportunities to strengthen your current code management processes.

- What third-party code is in use today?
- Can the specific owner and software version be identified?
- Where did it come from?
- When was it obtained?
- How long has it been in use?
- What license(s) apply to the third-party code?
- Which version of the license?
- Is a copy of the license on file?
- Does the license permit commercial use?
- Does the license contain copyleft terms?
- Does the license have any clauses that raise patent issues? (Licensing, indemnification, existing infringement suits, etc.)
- What is the intended usage scope for the third-party code?
    - Is the proposed use internal, or will it be incorporated into product(s) distributed outside your organization?
    - Will the third-party code be used in the context of “software as a service” (client / server scenario)?
    - Will the third-party code be “pushed” to a user, such as JavaScript files to a user’s browser?
    - If distributed outside your organization, is the distribution under a commercial license or an Open Source license?
    - How will you comply with license requirements such as attribution notices or availability of your modified source code?
- Where and how is the third-party code used in your software?
    - If the third-party code is closely integrated with your own development, how do you tell who owns what?
    - What are the license terms that govern copyright ownership?
    - If there are multiple owners, do they jointly own everything?
    - If not, what are the rules that govern ownership?
    - Can you apply these rules to specifically identify what you own?
        - If not, you need to clarify the rules
        - If so, generate a list of the specific programs (“manifest”) for which you claim copyright. Generate a separate manifest for jointly owned materials. Generate a third manifest for the parts you do not own.
- Has the third-party code been reviewed to identify whether it is subject to known security vulnerabilities?
- Are all license compliance requirements related to the third-party code met?
    - For developers?
    - For distribution to end customers?
- What processes are in place to track compliance issues?
- Who is responsible for compliance?
- Who is responsible for code review?
- Are the guidelines used in the code review process clearly documented?
- What directives have developers been given with respect to the use of third-party code?
- Have developers affirmed they understand and follow these directives?
- What processes ensure developers are following the directives?
- Are there published guidelines for developers who contribute to Open Source initiatives?

- Once third-party code has been used, what procedures govern its re-use for subsequent development projects?
    - Are your developers aware of the specific procedures?
- Are there controls over third-party code use? (Example: developers may be required to “check out” third-party source from a centralized repository of approved code)
    - Do these controls properly address re-use of code in future development?
    - Do guidelines clearly distinguish between code that can be used internally and code that can be redistributed?
- Do you plan to perform periodic compliance audits?
    - If so, who will perform the audit?
    - What process will be used?
    - How will the audit results be documented?
- If an audit reveals the need for remedial action, do you have a plan?
    - Who will decide what action is appropriate?
    - Are there findings that might require you to stop shipping product?
    - What criteria will be used to make this determination?
    - If so, what notification procedures will be used, both internally and externally?
    - Does the process differ if there is a licensing problem vs. security vulnerability?
    - What will you do if you discover a “back door” in your code?
    - Who will perform the required actions, which may require involvement of both legal and technical participants?
    - Who pays for the remediation?
- If follow-up is required post-audit, who will be responsible for the follow-up?
- What steps have you taken to ensure that your developers and consultants do not incorporate code written for past employers into your products?
- Have you communicated your code management standards to your consultants, contractors, and software development partners?
- Does your business plan include success based on longer trajectory of time or shorter? In either case, if code management is not a priority is there a budgeted ‘war chest’ for the costs of emergency and expedited management? (In many ways, code management is akin to changing the oil in your car. You can avoid it, but the avoidance will be costly.)

Inspiring, Enabling and Driving Quality Improvement

Jim Sartain
jsartain@adobe.com

Abstract

This paper discusses the approach used at Adobe Systems for driving continuous quality and engineering process improvement through the adoption of engineering best practices including Team Software Process (TSP), Scrum, Peer Reviews and Unit Testing. It covers what has worked well and some challenges encountered. Key success factors and an overall methodology for driving improvement are outlined including:

- A multi-year improvement approach that can work in large organizations that may include both eager adopters and resisters of process improvement.
- Customer Satisfaction and Defect Removal metrics, including what is world class, what is typical, and the best practices that can drive major quality and engineering process improvement.

Biography

Jim Sartain is a Senior Director responsible for Software Quality at Adobe Systems (makers of Photoshop, Acrobat, and other industry leading technologies). He leads a team responsible for inspiring, driving and enabling continuous quality improvement across Adobe world-wide. Prior to Adobe Systems, Jim held quality/engineering process improvement and software development leadership positions at Intuit and Hewlett-Packard. While at Intuit, Jim drove significant improvement in quality and software engineering practices across the company. 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 Company Background

Adobe Systems is a leading provider of solutions that enable customers to create rich and engaging internet experiences. Headquartered in San Jose, California, Adobe had close to $3.0 billion in revenue in fiscal 2009 with greater than half of this revenue generated outside the United States. In 2010, Adobe Systems was ranked #1 on Fortune Magazine's list of the world's most admired software companies.

Adobe has more than 8600 employees with software development in many countries including the United States, India, China, Canada, Romania, Germany and Japan. Popular products include Photoshop®, Premiere Pro®, Acrobat®, Dreamweaver®, InDesign®, Flash Authoring® and LiveCycle®. Adobe has some ubiquitous products including Adobe® Flash® Player and Photoshop®. Flash® is on 98% of connected PC's and has 8 million installs per day. More than 90% of creative professionals have Adobe Photoshop® software on their computers.

2 The Business Opportunity for Quality Improvement

Some software development teams spend a significant portion of their development effort on defect driven rework. Software development activities beyond when software is declared "functionally complete" – a point at which teams are primarily focused on finding and fixings bugs is typically rework. I've seen some teams at HP, Intuit or Adobe Systems consume up to one-third of their overall development effort on testing and fixing bugs with another 15%-20% of effort required post-release to deliver incremental "dot" releases that repair shipped defects.

Organizations employing quality-first software development approaches can reduce these rework costs to less than 10% of overall development effort. These teams deliver software on a regular basis (e.g. monthly) that is at or near release quality and these releases do not typically require subsequent dot releases to address quality issues.

Another significant driver of software quality costs is customer service 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 customer issues.

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

3 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 9 or 10 on a 0 – 10 scale. Promoters will tell their friends, families and business associates to buy and use the product and are responsible for generating up to 90% of positive product referrals. Customers rating their likelihood to recommend from 0 to 6 are considered to be detractors. Detractors are responsible for up to 90% of negative word-of-mouth. Social media (e.g. twitter, blogs) provide an increasingly visible means for product promoters and detractors to share their views with thousands of customers world-wide. A world-class NPS for commercial software products is in the 50% to 70% range.

This 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 % of customers that are promoters and subtract the % of customers that are detractors. Figure 2 illustrates this calculation.

Figure 2: Net Promoter Score Formula

4 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. Having this vision is important even for organizations that are doing well with quality Otherwise a natural tendency is to accept the current level of customer experience and organizational effectiveness as "good enough". The vision should include compelling benefits for key stakeholders of quality including customers, employees and shareholders.

4.1 Customer Benefits

The vision should emphasize directly engaging with customers utilizing techniques such as NPS to maximize the number of product promoters. Adobe's goal is to have 70% of its customer be promoters. Adobe has found that it's most satisfied customers:

- Are more likely to recommend Adobe products to others
- Invest a greater percentage of their budgets with Adobe
- Are less likely to consider alternative solutions

4.2 Employee Benefits

Employee benefits should include improved work-life balance. Also, employees will benefit from having more time available to do valuable work. Providing a less stressful work environment that provides more time to do truly innovative work is a great way to increase employee engagement and ultimately retention. Some Adobe teams have freed up 20% more time from reductions in defect-driven rework.

4.3 Shareholder Benefits

Improving product quality can increase the number of product promoters and decrease the number of detractors. Increasing promoters and decreasing detractors will lead to increased sales, customer retention and the ability to charge premium prices.

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.

5 Required Leadership Support

Most software engineers have a strong desire to deliver software of high quality and will if they are provided the right environment. This environment must include 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.

An important leadership responsibility is to ensure delivering quality software that meets business goals is the highest priority for the organization. Business goals will typically include the delivery of software on a specific 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.

5.1 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. Teams should have compelling quality improvement goals that include specific measurable goals and realistic improvement targets. Improvement targets should be set by the teams that own accomplishing them to 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. 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. For example, tying a goal to improve productivity to providing the opportunity to do more important and interesting work can be more relevant to employees and more likely to gain their support.

To help set aggressive but achievable goals it is useful to have benchmarks. 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.

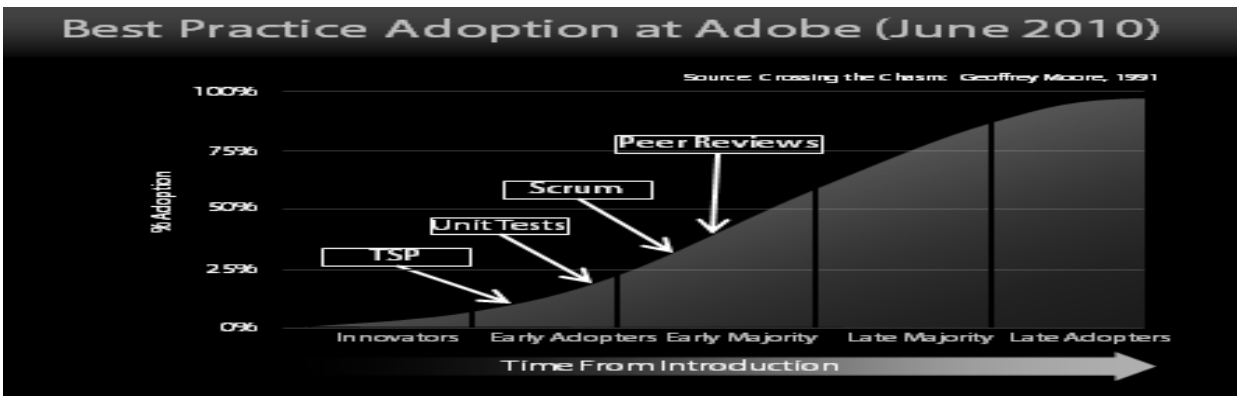
5.2 Quality Review Meetings

The senior leadership team must regularly review quality as a team. During these review meetings each team leader summarizes improvement progress and status. This provides an opportunity to recognize teams making progress. By publicly recognizing progress leaders reinforce the importance of continuous improvement. In cases where progress is not being made there should be discussion as to why and commitment and the 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.

6 Engineering Best Practice Rollout Strategy

Most engineering best practices have been around for years and are known to many engineers as best practices. Why aren't they common practice? Teams must take risk to do their work a new way. They must invest extra effort to learn new tools and techniques and usually need training and consulting on how to do them well. Best practices usually 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.

The level of motivation for adopting change normally varies from team to team, from highly receptive to resistant. 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. Adobe uses a viral approach where teams are encouraged to be pioneers. 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. Engineering best practice adoption at Adobe has followed a classic technology adoption curve as described by (Moore 1991). Figure 3 below illustrates the current level of adoption.



Scrum and the Team Software Process (TSP) are being used by teams to provide effective project management frameworks. Both methodologies have helped teams to become more effective at self-management as well as focusing on delivering quality code using iterative development. They both make use of early defect removal practices such as Peer Reviews and Unit Testing and tools for estimation and continuous learning from project team retrospectives. Based on the current rate of adoption, 80% of teams at Adobe will adopt either Scrum and/or TSP by the end of 2012.

Scrum is an increasingly popular methodology for managing projects using an agile development approach. A majority of teams adopting Scrum at Adobe demonstrated significant improvements in software quality, schedule predictability and employee satisfaction with their work process. Scrum is becoming the dominate and fastest growing approach to software project management at Adobe.

TSP is a methodology that provides training, support and tools to enable a team to be self-managed and to deliver on business objectives with exceptionally high quality and engineering productivity. TSP is described in Watt Humphrey's book "Winning with Software" (Humphrey 2001). TSP teams collect and use excellent software quality metrics to guide their projects. Further sections of this paper summarize results from Adobe TSP teams as an example of what is possible when quality plans and metrics are used to drive a quality-first strategy.

7 Metrics are key to Driving Effective Best Practice Adoption

A critical tool for driving engineering best practice adoption is the use of metrics to ensure effective use and to demonstrate their ROI. Figure 4 below shows a key effectiveness metric for defect removal best practices – the number of minutes of effort to discovery 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 hours.

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 not sufficient. Even if lots of 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 % 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.

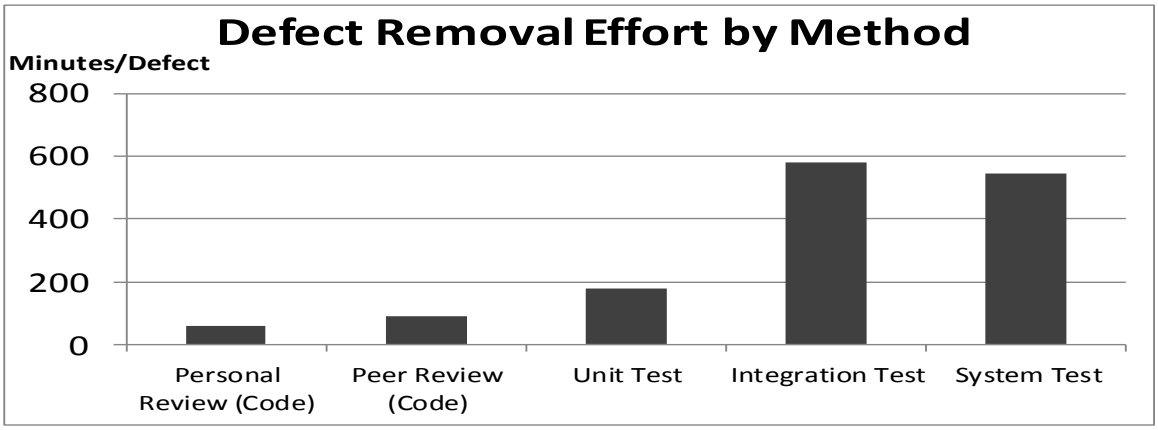


Figure 4: Defect Removal Effort by Method – Adobe TSP Projects

The data from Figure 4 was collected from Adobe TSP projects. Many Scrum teams are starting to collect this same data as a way to ensure these defect removal practices are being done effectively.

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 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).

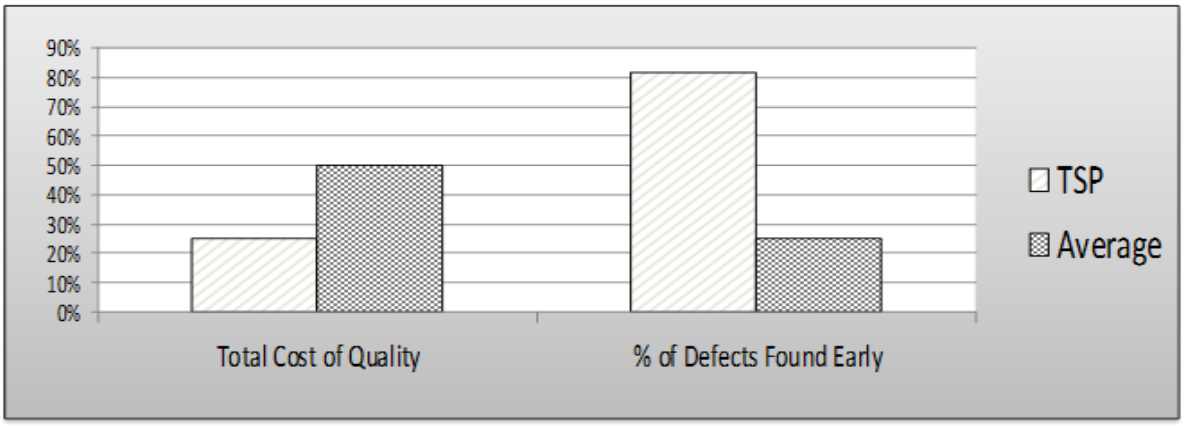


Figure 6: Team Software Process Early Defect Removal Results

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 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.

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 at Adobe 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. Also, I've learned so much about software engineering best practices and how to make this stick from Watts and Noopur. Thanks to Barry Hills, Johnny Loiacono, Paul Gubbay, Dave Burkett, Bill Hensler, Winston Hendrickson, Karen Catlin and Kevin Lynch for their strong support of the Adobe Quality Initiative.

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

Document Your Software Project

Ian Dees
ian.s.dees@tektronix.com

Abstract

Software projects are growing in complexity. We're expected to get more done in less time than we did last year. It's vital to come up to speed quickly on any given body of code, whether it's a new third-party library or a neglected legacy subsystem.

Many projects lean on API documents that have been automatically extracted from source code comments. Other teams churn out elaborate diagrams that explain every detail of the architecture. While these can both be helpful as a reference, they do little to answer the basic questions about the code: What's this for? What file should I look in first? What's this project's equivalent of a “Hello world!” program?

In this paper, we'll use the metaphor of a magazine article to think about the ways you can help anyone come up to speed with a code base, whether it's a new hire today or you, five years from now. Getting the emphasis and level of detail right are crucial. Everything else, especially the specific choice of tools, is secondary. Even so, we'll spend some time looking at a couple of open-source software packages that may be of help.

Biography

Ian saw his first Timex Sinclair 1000 over 20 years ago, and was instantly hooked. Since then, he’s debugged embedded assembly code using an oscilloscope, written desktop apps in C++, and joyfully employed scripting languages to make testing less painful. Ian currently writes GUI code for measurement instruments as a Software Engineer at Tektronix.

Ian is the author of Scripted GUI Testing With Ruby, and co-author of Using JRuby.

1. Introduction

“Welcome to the team. Here are a couple of bugs you can fix to learn your way around the source code. You’ll probably want to start by looking in this subsystem.”

Have you heard this before? Perhaps it was followed by a wild goose chase through the code, as you strove to read the mind of the coder who came before you.

Or perhaps it turned out differently. You may have opened the project folder to find a welcome mat of sorts. Someone may have carefully laid out a `README` file giving a quick overview of the directory structure, names of functions to look at first, and even code samples hinting at how to use or extend the code. Maybe that someone was you, several years ago.

In this article, we’re going to discuss how programming documentation goes wrong, and what you can do about it. It doesn’t take a world-class technical writer to document a software project (and it certainly didn’t take one to produce this paper!). It just takes a few times being baffled by an unfamiliar code base, plus a dose of empathy for your fellow developers. The number one question in their minds when they first pick up your library is, “Anything I need to know before I use this?” Let’s look at a few of the ways you can provide an answer.

2. What Do You Mean, “Documentation?”

Any kind of technical documentation can benefit from care and attention to detail. In-code commentary, user manuals, architectural walkthroughs, and requirements documents should all be written in clear language and kept up to date. On any given project, each of these types is going to come with its own set of constraints on length, level of formality, and assumptions about its audience.

This article focuses on one kind of writing: software library documentation. Your audience will be your fellow programmers, very likely including yourself. The scope is something quite a bit narrower than an entire system—something along the lines of one library or subsystem.

Even the small world of a single software library can contain a vibrant ecosystem of documentation: READMEs, code comments, API references, and more. Jacob Kaplan-Moss, one of the creators of the Django web framework, classifies project documents into three broad types:[1]

1. *Tutorials* give the new programmer enough information to get started.
2. *Topic guides* pursue one aspect of the project in great detail.
3. *References* provide exhaustive coverage.

It’s worth taking a look at your software project to see how you’re meeting all three of these needs for your readers. This paper will be dealing mainly with the first category: getting-started documentation for new users of your library. You want to lead your reader to an early success, then challenge him with a couple more realistic examples, then point him toward the more in-depth resources that live elsewhere.

Topic guides are crucial as your project’s user base grows. For that reason, they’re outside the scope of what we’re talking about today: how to jump into a new project (as a reader or writer) without getting bogged down. References are tangentially related to the discussion, so we’ll touch on them in a couple of places.

[1] http://jacobian.org/writing/great-documentation/what-to-write

2.1. Thought Experiment

Think back to the last time you visited an unfamiliar subdirectory of your source tree for the first time. Perhaps you were assigned to help another teammate develop a feature, or simply found an interesting name in the project directory and wondered, “What’s in here?”

How easy was it to learn the purpose of that particular subsystem? Was there a `README`, or was it cultural knowledge among the team what that directory was for? Was there any documentation at all in the directory, or were you expected to look in some central location for everything?

If you did find documentation, how helpful was it? Was it boring or engaging? Did it give a quick overview of the more important files in the directory? Was there a “Hello world” code example you could try out? If so, was the example up to date?

Keep your answers to these questions in mind as you read the next couple of sections.

3. Document Misfits

People write documents for good reasons. (And not just “because management said to.”) You can probably think of several right off the bat: pride in one’s work, sympathy for one’s co-workers, or even simple self-preservation.

What goes wrong, then? Assuming authors' hearts are in the right place, we can imagine that a document that fails us may just be the wrong kind of document for the task. Let’s look at a few ways in which a software library might not be getting the textual attention it deserves.

3.1. Invisible Ink

A subsystem might have no documentation at all. This scenario is distressingly common, and to my embarrassment I’ve been party to it as well. I’m going to take advantage of the bully pulpit afforded to paper authors, and speculate on some of the causes.

We skip documentation because we run out of time. We skip it because we’re self-conscious about writing narrative text. We start a tiny new project and think, “This thing’s way too small to bother documenting; just look at the source code!” We get frustrated with flaky writing tools and bureaucratic workflows.

As we’ll soon see, there are a few simple things you can do to address nearly all of these causes of missing documentation.

3.2. Ghost Writer

A close cousin of the invisible document is the ghost-written document; that is, the automated help pages generated from the declarations (and comments, if there are any) in your source code. It’s certainly fine to use a tool to help you generate finished documentation. But tools can’t create something out of nothing.

Kaplan-Moss views auto-generated documentation as “almost worthless. At best it’s a slightly improved version of simply browsing through the source….” This is a justifiable reaction to visiting a project’s website in search of documentation, and seeing nothing but a raw list of class and method names. Hand-written text, he argues, is much more useful to the reader.

The contrary point of view is that tools like Javadoc (or its equivalent for your language) can encourage you to do a thorough job commenting all the parts of your code: function behavior, correct use, gotchas, and so on. That’s exactly the kind of information a reader is looking for when he moves on from the tutorial in search of a reference guide.

In the spirit of unifying the two camps, consider the fact that most of these tools can import external hand-written text files into their generated documents. When you've spent hours writing a detailed overview of your project's API—and hours more explaining how to use every public method of every class—does that even count as automatically generated documentation any more?

A successful project maintainer views writing documentation as an intensely (pardon the pun) manual effort, no matter whether or not the end result happens to have been generated with help from a source tool. For an example of a document maintained separately from the source code, pick any given section of the Python standard library.[2] For an integrated document that pulls in some information from the source tree, see the reference for the Haml markup language.[3]

We'll return to the subject of source comments later on in the article.

3.3. One Document to Rule Them All

Some shops demand that one file or directory be the gateway to all understanding of the project. Hardware layouts, software architecture, UML diagrams, elaborate class hierarchies, and everything else gets dumped into this one location.

It's easy to see why this approach is so common. If every fact about the system lives in one place, then one can turn to the master document for any answer—in theory. The problem is that these would-be museums quickly become mausoleums. Those elaborate diagrams that got produced not out of any real need, but because someone thought the team should have them? Dumped in the burial ground and forgotten.

The problem is not just that we might have trouble finding a specific piece of information. After all, tools like `grep` and desktop search can help us get through cluttered directories. The problem is that there are pieces of information that we don't even know exist. Too often, I've struggled with some API because the advice on how to use it was buried in a folder I'd never seen.

The exception to this anti-pattern seems to be wikis. In theory, these should be just as susceptible to information imprisonment as any other centralized documentation storage. But in practice, I've seen successful project wikis that made it easy to retrieve project knowledge—not just answers to the questions I had, but also answers to the questions I didn't know to ask. Perhaps the ease of hyperlinked, ad-hoc structure is what makes the difference.

3.4. Ancient Scrolls

Documents need to be kept up to date. It feels silly even having to say this. But I've definitely encountered libraries whose example code no longer ran unaltered, or whose descriptions didn't match the API any more. I'll bet you have, too.

People don't come into the office and decide to put off documentation updates. Maintaining the docs just doesn't become part of our habits, for the same reason that creating them doesn't become part of our habits. Every little barrier in the way of writing looks like an impenetrable wall when we think about doing it over and over again in maintenance mode.

It follows that any little thing we can do to stop ourselves from dreading writing and just *write* will have a big payoff. I'd like to discuss several such things you can do in service of this goal. To tie these various activities together, I'd like to propose a metaphor for software documentation that carries warm, familiar connotations for most of us.

[2] http://docs.python.org
[3] http://haml-lang.com/docs/yardoc/file.HAML_REFERENCE.html

4. The Magazine Metaphor

When you were cutting your teeth as a coder, did you ever subscribe to *Dr. Dobb's Journal*? How about *BYTE*, or *COMPUTE!*? Remember the thrill of diving into some brand new technique, and being presented with real, live code samples that you could actually type in and run right on the spot?

It may be a bit unrealistic to expect users of our libraries to feel quite that level of excitement about software that's just part of our day jobs. Even so, a well-written project tutorial has a few things in common with a short programming article in a magazine. Let's look into a few of these features.

4.1. One Sitting

The engineer trying out your library doesn't have all day to read an entire binder's worth of writing. There's a place for exhaustive coverage of a subsystem: the reference material. Instead, give your reader a few pages that can be printed out and read over a cup of coffee.

4.2. Working Examples

One of the most heavily leaned-on subsystems in our office has a well-loved usage guide. Even though the document doesn't score 100% on all the counts we've discussed (e.g., being easy to find and up to date), it's always worth the effort of tracking it down and reading it—because it does such a thorough job explaining how to write code for this subsystem.

In one sixty-line example that fits on a single page, the author is able to demonstrate all the library's major features. In the adjoining text, he explains what each single line of sample code does. After refreshing our memory of how that one example works, we usually have enough information to jump right in and code. For those rare cases where we need a little more, he's provided a thorough API reference in the back.

The best thing about this document is that it's like a journey through the mind of its author. Understand him, and you understand the code.

4.3. Curated Code Excerpts

The most enjoyable articles show more than just one "Hello world" example. They compare and contrast what code might look like before and after applying the technique they're describing. They peel back the API and show some facet of the underlying implementation.

I sometimes struggle with how much code to show in an introductory document. Too little, and the reader won't know where to turn after getting the first example working. Too much, and the whole thing degenerates into a parade of excerpts that don't have enough context to be meaningful. This is where the "magazine article" theme comes in most helpful to me. Articles show off a small series of curated code examples that capture the essence of a library.

4.4. A Real Page-Turner

With all this talk of keeping things short, you're probably wondering how to fit in everything the reader might need to know about the software. It's important to provide good coverage, but an introduction is not the place for that. The introduction absolutely *is* the place, however, for telling your reader where to get more information: a series of in-depth guides, full API documentation, a network of wiki pages, and so on.

5. How To Get There From Here

There's no need to go overboard with the magazine metaphor. You don't need fancy formatting, callouts, author bios, personality quizzes, or intrusive ads. Simply use the idea insofar as it helps you get the job done, and discard it as soon as it loses its usefulness.

Assuming you do find a few parts of the idea useful, how can you incorporate them into your workflow?

5.1. Keep It With the Code

The further you have to reach from your source tree to maintain your documents, the less often you're going to maintain them. It's fine to have a central place for the big stuff: giant architecture diagrams (on second thought, maybe you could just do without those), requirements documents, histories of design decisions, and so on. But your readers need *something* right next to the source code to get started—and so do you.

You may be thinking, "But we don't want to clutter up the source tree! There's no conceptual room for a giant `README`." Precisely. Keeping this material right next to the source should provide an upper bound on its size and complexity.

5.2. Use Tools That Fit Your Hand

We're all under schedule pressure. How do we make sure that the task of creating and maintaining the documentation isn't the first thing thrown out the window as the deadline draws near? Choosing a lightweight format is one way we can trick ourselves into doing the right thing. If we think of our project guide as a magazine article, it seems less difficult and more approachable.

Choosing a writing tool is another part of this psychological game. Use the tool that's not going to become an excuse for you to skip out on your writing duties. If you find yourself thinking, "Now I have to wait all day for the word processor to launch," you may prefer a plain-text format and the comfort of your favorite programming editor. If the prospect of memorizing obscure bold / italic codes annoys you, a dedicated writing tool may be the best fit.

5.2.1. Plain Text and Its Relatives

Don't underestimate the power of plain text. It can be read from the command line with no additional tools, sent around as e-mail, and tracked usefully with revision control tools. This includes the "humane markup" languages, such as Textile, Markdown, reStructuredText, and AsciiDoc. Using one of these formats means just writing like you'd normally do, and then adopting a few simple conventions for section titles, code excerpts, and so on. For example, here's how a README file might begin in reStructuredText: [4]

```
Halt-o-Meter
============
Welcome to the Halt-o-Meter! This software reads the source code of
your program and tells you whether or not it will run to
completion (and they said it was impossible!).

Save the following code in ``example.c``::

  int main() {
    for (;;);
    return 0;
  }
```

[4] http://docutils.sourceforge.net/rst.html

```
Now, run Halt-o-Meter::

  C:\> haltometer example.c
  Checking... example.c does NOT halt.
```

The other text formats look similar to this. I chose reStructuredText for this example because it's the one used by Sphinx, the tool behind the Python library documentation. [5] Sphinx is designed for writing books and articles, so it's very close to the sweet spot for introducing readers to a software library. If you happen to be writing about a Python project, Sphinx can link to your generated source documentation.

5.2.2. Machine-friendly Markup

Formats such as DocBook, LaTeX, HTML, and RTF are also text-based, but typically involve inserting instructions for the typesetting among the words of your document. To one degree or another, they're more difficult to write by hand than plain text (LaTeX is the easiest in this regard). This added complexity pays off, though, when it's time to ask a machine to do something to your text—such as print it, translate it into another format, or automatically update snippets of your source code.

Incidentally, the AsciiDoc format described earlier generates DocBook XML behind the scenes.[6] So you can get the best of both worlds: the ease of plain text, and the feature set of the entire DocBook toolchain.

5.2.3. Word Processors

The tools we've talked about so far have been pretty programmer-oriented. A few of my teammates prefer a change of scene when they're working on documentation. They'd rather see formatting changes while they're interacting with a document, rather than having to wait for a conversion to PDF.

So they fire up a word processor and start writing. They use styles and templates to avoid getting mired down in micro-decisions about the document's look. And they produce readable, useful documentation.

One thing to watch out for if you go this route is that you're going to have to work a little harder to keep your code examples up to date.

5.3. Keep the Code Up to Date

Having stale code examples that break for your first-time users is a recipe for frustration. If your library implements a fairly stable API, especially one that's defined in a standard somewhere, you may not have to worry about this much—just go ahead and paste your source examples into your document.

But if your project's programming interface changes frequently, you'll need to re-run your examples and keep the versions inside your document up to date. This task is easier with the text-based formats; most have commands that say in effect, "insert the source code from this external file here." For the ones that lack this feature, you can use a templating engine such as the Ruby-based eRubis or the language-independent m4. [7] [8]

Automatic updates are a little trickier when you're using a word processor. If you're only responsible for documenting a couple of projects, you might just add a recurring task in your calendar to try the snippets manually and make any needed fixes. Any more than that, though, and the manual approach will get old quickly. You can use your word processor's linking feature and store your snippets in external files (which are easier to test automatically).

---

[5] http://sphinx.pocoo.org
[6] http://www.methods.co.nz/asciidoc
[7] http://www.kuwata-lab.com/erubis
[8] http://www.gnu.org/software/m4/m4.html

You can also just update the document by writing a program. Most word processors have either a scripting API or an XML-based file format that's easy to generate automatically. [9]

5.4. Provide a Path from "Hello world" to Mastery

An engineer who stumbles across your project directory has a number of questions, which almost certainly include the following:

- What is this thing?
- How do I get it and all the dependencies installed?
- What's the equivalent of "Hello world" for this system?

Kaplan-Moss reminds us to "Be easy…. But not too easy." You want your reader to get an early success, then challenge him with a couple more realistic examples, then point him toward the more in-depth resources that live elsewhere.

6. Real Life

All this talk of misfits and magazines is all well and good, but how do these ideas apply in the real world? Do real projects use these techniques? What specific tools come in handy? And how are we software developers supposed to find the discipline to write?

6.1. An Achievable Example

Jumping into real-world examples carries a certain amount of risk. Reactions to any featured project will range from, "How can you praise the documentation for *this*?" to "You should have covered Project X instead!" Choose too many examples, and the whole discussion devolves into a laundry list.

With that in mind, let's shoot for a realistic example. We don't need to see the most beautiful, exhaustive documentation in the world. We need to see the kind of thing that you or I could reasonably produce on a tight time budget.

The tutorial for the Sass stylesheet language fits these criteria.[10]. Sass, by the way, makes it easier for web developers to implement Cascading Style Sheets (CSS) designs. (I have no connection to Sass, other than as a user.)

Sass's tutorial flows like a magazine article from an old coding journal. It starts with a simple "Hello world" equivalent, then demonstrates a few features in the order that readers are likely to wonder about first. It mentions more advanced options in passing, but wisely stays on track. You can work your way through the whole tutorial over a lunch break.

Many of the hints in the tutorial link to deeper coverage on the library.[11] This next level isn't quite the exhaustive feature reference from Kaplan-Moss's hierarchy of documentation. But it does provide hand-written discussion, options, and examples for all the major features.

The reference also contains a list of classes and methods, but doesn't leave you at their mercy. The narrative material contains several links to the classes you're likely to need to know about first when you're slinging code. The method descriptions may have been converted to HTML with the help of a tool, but they were written by a human being.

[9] For a related example of updating a presentation programmatically, see http://github.com/undees/snippetize.
[10] http://sass-lang.com/tutorial
[11] http://sass-lang.com/docs/yardoc/file.SASS_REFERENCE.html

How did the Sass team get this documentation done? They put it right into the source tree! The text is less likely to fall out of date than if it lived in some separate location. It's easy for project newcomers to view and contribute to the documentation, because it's in Markdown format and can therefore be read and written without any special tools.

6.2. Speaking of Code…

Earlier, we touched on keeping documentation with the code. In some cases, this means keeping parts of it *in* the code. It's possible to generate API documentation from source files. But it's also easy to over-rely on tools, and end up with a document that's not much more help than just reading the source code.

What level of detail is appropriate for documenting a class or a function? No single answer will suit all projects. Here are a couple of general themes that have been true on most of the project teams I've been on.

First, we have to start somewhere. Don't let the fact that some legacy class has 30 methods stop you from documenting the first one. Just think of the answers to a few simple questions about the file you're editing. Which function will project newcomers need to call first? Which one do experienced developers frequently get wrong?[12]

Second, shy away from those giant comment templates that have required sections with names like `NAME`, `DESCRIPTION`, `PURPOSE`, `ARGUMENTS`, and `REVISION HISTORY`. Your tools should pick up the function's name and parameters from its argument list.[13] Instead, just write free-form text. That'll keep you writing, and it'll save your readers from having to slog through parameter descriptions like "second_number: the second number to be added."

If we return to the example of the Haml project, we see that their source code for public APIs contains these kinds of comments. Consider their `Filters` module.[14] The file begins with a three-paragraph description, complete with a simple example. By opting for something more than just a single sentence, they've deftly avoided the trap of writing things like, "The `WidgetManager` class manages widgets." They've also steered clear of the one-size-fits-all approach. Some functions are simple enough just to need a two-liner. Others get the full treatment, with usage examples and discussions of valid parameter values.

6.3. Story Time

You've no doubt noticed that I've made it most of the way through this article without advocating any specific tools. That's by design. Not only is every developer different, but every project has its own needs.

So rather than recommending one set of tools for all situations, I'd like to tell the story of one situation, and how the choice of tools fell out of those circumstances.

A few months ago, our software lead and I were designing a new feature. We were in uncharted territory: we barely even had words for the things we were talking about. And we were supposed to be writing the requirements document for this feature.

We grabbed a conference room, sketched on whiteboards, and typed madly into our laptops. At this point, there was no conscious decision to use a particular format or processing tool. We were just using the tools that leapt into our hands first: the same programmer's text editors with which we write our code.

As the document took shape, a writing style emerged. We were subconsciously using the same kinds of text conventions we'd used for years for e-mail: dashes as crude underlines for section titles, asterisks for bulleted lists, and so on.

[12] This is also a good way to identify parts of your API that need to be redesigned. If people frequently confuse the fifth and sixth integer parameters to your function, perhaps the function should take fewer arguments.
[13] Dynamically typed languages may need a few extra hints.
[14] http://github.com/nex3/haml/blob/master/lib/haml/filters.rb

Once we were done with capturing all those thoughts, it was the work of a few minutes to transform the ad-hoc text format into one of the “humane markup” languages described earler. In this case, I chose the AsciiDoc format for its ease of making PDFs.

Because AsciiDoc generates DocBook output, any DocBook toolchain can be used to make the PDF. I happen to like Remko Tronçon’s DocBook Kit, a toolbox combining a nice set of styles with a Makefile that downloads all the processing tools for you.[15] It’s the work of a moment to grab a fresh copy of this kit, extract it into your project directory, adapt the sample Makefile, and type `make pdf`.

When we first showed the document to our teammates, we wondered if we’d catch flak for the fact that we weren’t (yet) using the official company documentation template. We don’t mind using those sorts of things, by the way—but there’s a time and a place for everything.

To our delight, people appreciated the content of the writing. They said that having this kind of “theory of operation” helped them understand the feature better than a fill-in-the-blanks template would have. Of course, when we reach the state of the project where people will expect the templates, we’ll preserve the narrative text.

6.4. The Discipline to Write

Stories about other other projects are all well and good, but they can only go so far. At some point, you’ll be picking up your own keyboard and writing. And that’s something that all the exhortations, analogies, tools, techniques, and testimonials can’t help you with.

How do we as developers cultivate the discipline to write? Advice along these lines usually amounts to tautology. “The only way to write is to write,” goes the line of thinking. This is sort of like saying, “The only way to diet is to eat less.” Well, sure, but there are ways dieters and writers can help themselves.

For me, it comes down to removing excuses not to write. If I feel intimidated by the prospect of writing a huge document, impatient with word processors, and lost looking for templates, that gives me three chances to put off writing and go do something else.

That’s why the main thrust of this argument has been: think of documentation in a way that makes it seem less daunting, choose tools that don’t give you any excuse to stop using them, and keep your documents close by. If you do that, I’ll be happy—not least because I’ll find a well-explained project if I should ever have the good fortune to work on a project with you.

7. Closing Thoughts

If you’re the maintainer of a small- to mid-sized software project, the programmers using your software deserve great documentation to help get them started. One day, one of those programmers may be you! For your sake and theirs, treat yourself to good documentation.

It’s easy for writing to get lost in competition with other, more urgent, activities: testing, coding, debugging, planning, and so on. You’ve got to get rid of any excuse not to write. That means choosing whichever tool is going to stay out of your way. It means setting realistic expectations of what you’re trying to accomplish.

If it helps, by all means choose a mental picture to rally behind. You may find inspiration in the old computing magazine articles that once inspired you to try out someone else’s code.

[15] http://github.com/remko/docbook-kit

THE LAST 9% UNCOVERED BLOCKS OF CODE – A CASE STUDY

Cristina Daniela Manu
cmanu@microsoft.com

Pooja Nagpal
ponagpal@microsoft.com

Donny Amalo
donnya@microsoft.com

Roy Patrick Tan
roytan@microsoft.com

Abstract

Code coverage is an efficient technique for understanding what code has been exercised by an existing test bed, yet it is often debated how much effort should be invested in increasing coverage. This paper is a case study of our code coverage effort during the product development cycle for a component of the .NET Framework 4. At the end of the cycle, we had a focused initiative to increase the code coverage from 91% to 100%. In order to measure the effectiveness of increasing code coverage, we devised a few metrics based on the number and importance of the bugs found, the time invested, the increase in code coverage and bug yield in comparison with other test activities. Our results showed that:

1. It is not prohibitively expensive to achieve effective[1]-100% code coverage.
2. There is no significant increase in the number of bugs found with higher code coverage.

In this paper, we discuss what parts of our development process (such as the propensity of testers to develop more complex scenarios first) may have contributed to this lack of bugs.

Biography

Cristina Manu is a Software Development Engineer in Test at Microsoft, in the Technical Computing group, working on testing libraries for supporting parallel applications. Before joining Microsoft, she was a lecturer at "Politehnica University of Bucharest" teaching mathematics. She holds a PhD degree in mathematics from University Of Bucharest.

Donny Amalo is a Software Development Engineer in Test at Microsoft in the Technical Computing group working on testing libraries for supporting parallel applications.

Pooja Nagpal is a Software Development Engineer in Test at Microsoft in the Technical Computing group working on testing libraries for supporting parallel applications.

Roy Tan is a Software Development Engineer in Test at Microsoft, in the Technical Computing group, working on testing libraries for supporting parallel applications. He earned his PhD in Computer Science at Virginia Tech in 2007.

---

[1] We define "effective code coverage" as the coverage of all the reachable blocks of code.

1. Introduction

Towards the end of the product development cycle for the Parallel Extensions to the .Net Framework 4 (7), our test team was able to reach 91% code coverage. We wanted to investigate what was left in the uncovered areas to make sure we added tests for it. Further, we wanted to understand the test investment required for increasing code coverage from 91% to 100% and what benefits would there be beyond discovering new bugs.

Managers often struggle with the decision about how many resources to allocate for increasing code coverage values. Several studies (e.g., (1) (2) (3)) have claimed that code coverage is a good technique to measure test effectiveness. In fact, several studies claim that test suites with higher code coverage have higher bug finding rates. A study by Hutchins et al. (1), for example, found that as code coverage goes from 90% to 100%, test suites get exponentially better fault detection rates. Similar work by Frankl and Iakounenko (4) and Andrews et al. (5) also claimed exponentially increasing bug finding ability as code coverage gets closer to 100%.

On the other hand, some research has also found that increasing code coverage is prohibitively expensive, as in the same study above by Andrews et al. (5), and including research done at Microsoft and Avaya (6). This paper is a case study of our code coverage efforts to cover the last 9% of uncovered code.

We show that increasing code coverage from 91% to effectively 100% need not be prohibitively expensive. However, we also show that find increasing code coverage did not find an exponentially increasing number of bugs.

The paper is structured into the following sections:

1. Description of the project and of the test strategies used during the product development cycle.
2. Description of the metrics used for measuring the code coverage.
3. Description of the code coverage efforts.
4. Results and Conclusions

2. The Project

Parallel Extensions to .NET Framework 4 (7) is a set of library additions to the .NET Framework 4 that simplifies concurrent and parallel programming. It consists of three main components:

Task Parallel Library (8) – this provides a rich task-based model for asynchronous and parallel programming. It defines continuation patterns and allows for custom scheduling and task-attachment relationships, while providing support for efficient joins along with cancellation and exception handling models. It also provides Parallel constructs including Parallel.For, Parallel.ForEach, and Parallel.Invoke.

PLINQ (9) – this is a parallel implementation of LINQ to Objects (10);

Data Structures for Coordination (11) – these are a set of important types that round out the Parallel Extensions, supporting necessary patterns and practices for adding parallelism to your applications. Important groups include thread-safe collections and synchronization primitives.

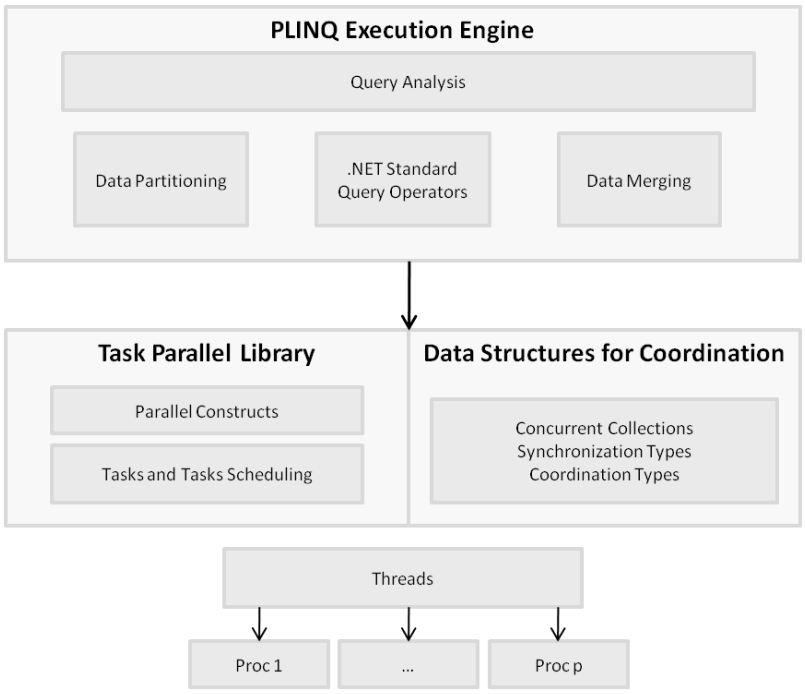


Figure 1:.NET Framework 4 Parallel Extensions Architecture

Classes	405
Functions	2397
LOC	55728
Blocks of Code[2]	16889

Table 1: Size of the Parallel Extensions product code

3. Test Strategies used for getting to 91% code coverage

Within the Parallel Extensions team, tests are developed simultaneously with the product code. These tests are a combination of tests written by the developer and tests written by the tester. By the time the product code is checked in, a single run of the test suite would have 100% pass rate. However, testing does not stop at the initial check-in of the product code. More thorough functional tests are typically developed as the product development cycle continues. In addition, we develop many other types of tests, including performance, stress, and security tests. For the purposes of this document, however, we only measure the code coverage [3] of *functional* and *fault injection* tests.

The functional tests employed by our team are tests that verify the intended behaviors of a software component. The functional tests comprise of a mix of manually written and automatically generated unit

[2] A block, also known as a basic block, is a set of contiguous instructions (code) in the physical layout of an executable that has exactly one entry point and one exit point. Calls, jumps, and branches mark the end of a block. A block typically consists of multiple machine-code instructions.
[3] Block code coverage numbers were captured through our in-house code coverage tool based on Vulcan binary analysis framework (14)

tests. Tests meant to check the integration between our components are also considered part of the functional test suite that gets run every day. In a typical test run, we have approximately 50,000 pass/fail results for our functional tests.

In addition to checking functional behavior, we also need to check whether our components behave correctly when faced with certain catastrophic failures. For example, our software component can be hosted in an AppDomain[4] , which can be unloaded at any time by the host application. We need to make sure that our software component shuts down gracefully in such cases (i.e., the software does not orphan threads or throw inappropriate exceptions). We check this behavior using tests that forcefully inject faults during execution. We identified approximately 50 such scenarios that were tested on a weekly basis.

As part of our test process, we occasionally have dedicated quality "pushes," where the test team focuses on a certain type of testing effort. Examples included a stress push, a performance push, etc. The data in this document is the outcome of two code coverage pushes. These pushes occurred towards the end of the product development cycle. At the start of the code coverage efforts, we already had 91% code coverage; the objective was to obtain 100% code coverage.

4. Measurements

To measure the value of test activities during certain time period, we formulated the following metrics:

Return of Test Effort (RTE), the sum of bugs found per day weighted by severity. In our case, the weight increased exponentially with the severity of the bug.

$$RTE = \frac{\sum_{k=1}^{maxSeverity} BugsOfSev(k) * (2)^{(maxSeverity+1-k)}}{TimeInvested}$$

Where:

TimeInvested = time invested per person

BugsOfSev(k) = the number of bugs of severity[5] k found during the current time period

Return of Coverage Efforts (RCE), the number of new blocks covered per day

$$RCE = \frac{\Delta(oldCC, newCC)}{TimeInvested}$$

Where:

Δ*(oldCC,newCC)* = currentCoveredBlocksValue – previousCoveredBlocksValue

Bug Finding Efficiency (BFE), the ratio of the number of bugs found by a specific test activity compared to other test activities during the time period

$$BFE(timeInterval) = \frac{CCBugCount(timeInterval)}{TotalBugCount(timeInterval)}\%$$

[4] An AppDomain is a feature that allows .NET programs to have isolation boundaries within a process.
[5] Severity of a bug is represented in our case by an integer value (between one and four) with one being the highest. A bug of severity one is usually a blocking bug or a high impact bug that needs to be fixed in maximum 24 hours.

Where:

CCBugCount(timeInterval) = the number of bugs found by code coverage activities during the time interval

TotalBugCount (timeInterval) = the number of bugs found by all test activities during the time interval

5. Results

We tried to cover the remaining 9% by breaking down the work into two test exercises specifically targeted to increase the code coverage numbers. We will refer to these two exercises as **First Code Coverage Push** and **Second Code Coverage Push**.

5.1.1 Goals

The **First Code Coverage Push** was focused only on *interesting* test holes. A test hole was considered to be *interesting* if it had one or more of the following characteristics: an uncovered block with high cyclomatic complexity (12), code that involves direct calls to public APIs, or code paths along core customer scenarios.

The main goal of the **Second Code Coverage Push** was to achieve coverage of all the blocks left uncovered after the First Code Coverage Push. Our approach was as follows:

1. Review each block of uncovered code.
2. Add the tests necessary to cover all the reachable code paths. If any path could not be reached, document the reason.
3. Identify the dead code and advocate having it removed.

The two code coverage efforts were done four months apart; during this interval a number of 136 more test cases were added and 334 were removed without any change in the overall block coverage value. The state of the test bed before we started the two code coverage pushes is summarized in the tables below.

#tests	block coverage	uncovered public methods
3649 test cases	91%	2.3%

Table 2: Data before the First Code Coverage Push

#tests	block coverage	uncovered public methods
3537 test cases	92%	0.2%

Table 3: Data before the Second Code Coverage Push

5.2 Collected Data

In both code coverage pushes, we collected the necessary data to calculate the RTE, RCE and BFE using the formulas defined in the Measurements Section. The data is summarized in the tables below.

code coverage push	#tests	block coverage	#of new covered blocks	uncovered public methods
#1	3735 test cases	92%	169	0.6%
#2	3829 test cases	97%	845	0.1%

Table 4: Tests and coverage at the end of each push

The primary reason we could not achieve higher than 97% coverage is the presence of dead code in our product. For a detailed explanation of the uncovered blocks, refer to section 5.4.

Note that the "#tests" in Table **4** is not equal to the difference between the tests before and after the code coverage push. This is because some tests were deleted during other test activities.

code coverage push	# bugs (severity 1)	# bugs (severity 2)	# bugs (severity 3)	# bugs (severity 4)	time invested (person days)	RTE
#1	0	4	0	0	10	3.2
#2	0	4	0	0	22	1.5

Table 5: Calculation of RTE

#Bugs in each column are the total of

1. bugs found by review of uncovered code during the code coverage push
2. bugs found by new tests during their lifetime (during and after the push)

code coverage push	∆ (oldCC, newCC)	time invested (person days)	RCE
#1	169	10	16.9
#2	845	22	38

Table 6: Calculation of RCE

BFE gives us an idea of the percentage of bugs found by code coverage activities compared to the total bugs found. To calculate BFE, we only look at the bugs found during the time period of the code coverage push.

code coverage push	total bugs found	bugs found by code coverage activities	BFE
#1	29	3	10.34%
#2	7	1	14.2%

Table 7: Calculation of BFE

From the table above, we can see that a large number of bugs are found by test activities other than code coverage investigations. To understand the value of each test activity, we categorized these bugs based on the activity that found them.

code coverage Push	customer reported	ad-hoc testing	code review	code coverage	regression	security	stress
#1	8 (27%)	7 (24%)	7 (24%)	3 (10.34%)	2 (6.89%)	1 (3.44%)	1 (3.44%)
#2	0	3 (42.9%)	0	1 (14.2%)	3(42.9%)	0	0

Table 8: Bug classification based on test activities

5.3 Analyzing the Data

To understand better the value of the two pushes, we looked at the proposed metrics: RTE, RCE and BFE in addition to the Bug Distribution.

5.3.1 RTE

In order to understand if the RTE from the code coverage pushes is good or not, we compared the combined code coverage RTE of the two pushes to the overall RTE of the product development cycle. The RTE for overall product development cycle using the same RTE formula defined in the Measurements section is **2.7.** This value is calculated using the time for overall product development cycle that include other test activities, like infrastructure architecture and development, test documentation development and review, etc. in addition to bugs finding activities and test case development.

# bugs (severity 1)	# bugs (severity 2)	# bugs (severity 3)	# bugs (severity 4)	time invested (person days)	RTE
0	8	0	0	32	2

Table 9: Calculation of Combined RTE of the two code coverage pushes

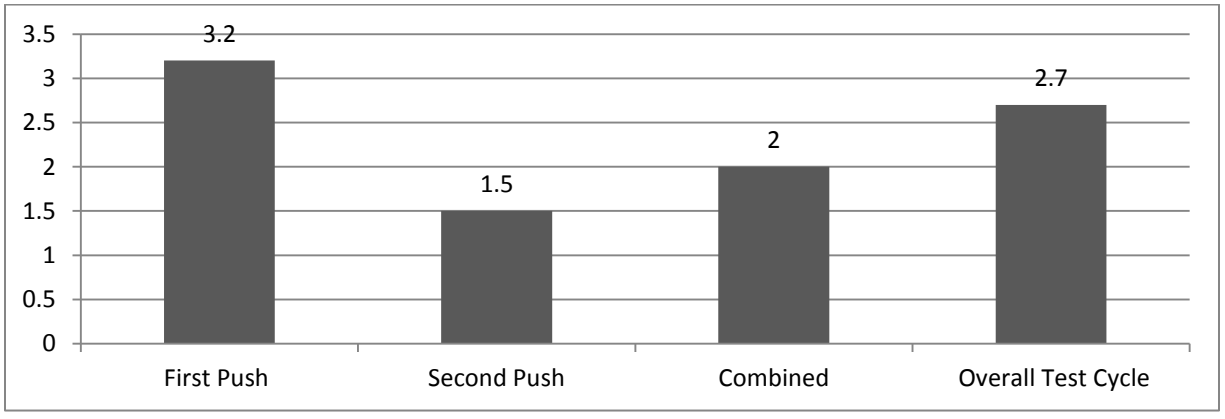


Figure 2: RTE comparison

From the results above we can conclude that the return on investment of the combined code coverage push is not as high as the rest of the product development cycle. However the first coverage push is somewhat better than the overall test cycle.

5.3.2 RCE

The RCE values calculated using the formula defined in Measurements section represent an estimation of the blocks covered per day. We use this value to estimate how easy to cover the last percentage of uncovered blocks is in comparison with the previous ones.

∆ (oldCC, newCC)	**time invested (person days)**	**RCE**
1014	32	31.6

Table 10: Calculation of Combined RCE of the two code coverage pushes

We compare these values with the RCE for the overall product development cycle. The RCE for the overall product development cycle is **6.7**. The comparison graph below shows that a focused code coverage push yields a higher coverage despite the fact that it is executed over the last blocks of code.

An explanation of this behavior might be the fact that during the regular product development cycle testers usually do not focus only on increasing the code coverage values. They execute a broad set of testing activities in order to assure a high quality test bed that will support the current and next product's versions.

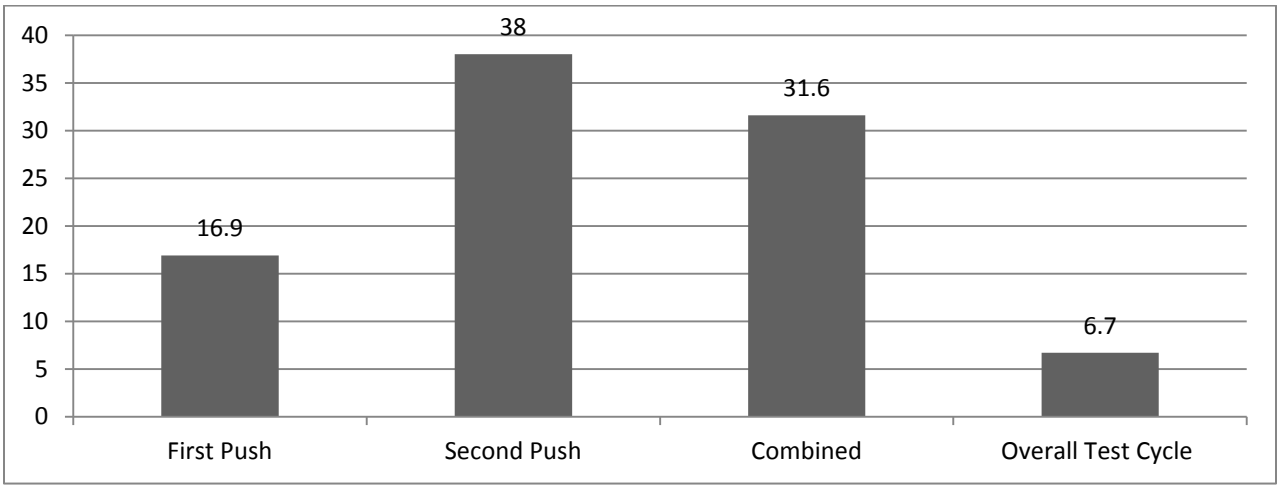


Figure 3: RCE comparison

5.3.3 BFE

BFE was a measure to understand how effective the code coverage activity was for finding bugs when compared to other test activities. We have only compared the efficiency of the two coverage pushes to understand which code coverage strategy was better in terms of bug finding capability.

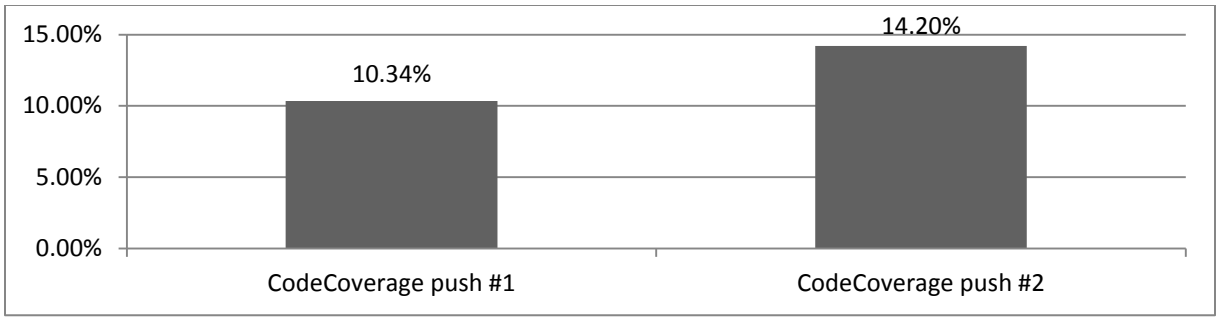


Figure 4: Code Coverage BFE

5.3.4 Bug classification

We studied the accumulated bug distribution from the two pushes to understand the value of the code coverage push when compared to other test activities. We want to highlight that this distribution is specific to the end of our product development cycle and may have been different at different points in the product development cycle. Our goal was to understand what test activities are effective when we already have high code coverage.

customer reported	ad-hoc testing	code review	code coverage	regression	security	stress
8	10	7	4	5	1	1

Table 11: Accumulated Bug classification based on test activities

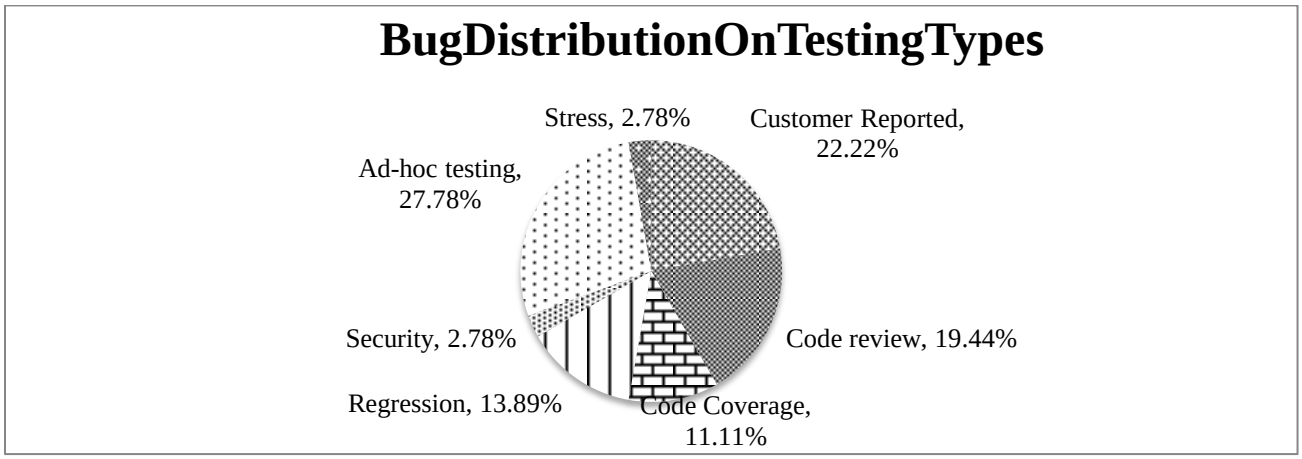


Figure 3: Bug Distribution

5.4 Why is it hard to achieve 100% coverage?

This is a general question and the answer can adjust for different products. Our purpose was to document our experience hoping that some of our observations will be applicable to other products. Below are enumerated the reasons that affected our efforts of achieving 100% code coverage.

1. *Dead Code*

Dead code should not be left in a product. Any code identified as “dead” should be removed for improving code maintainability. Removing the dead code should not have any effect over the product’s behavior if done with care and while removing it no other code paths are affected. For large products any code change requires a test pass because new bugs may be inadvertently introduced in the product along with the changes. At the time of writing this paper, our product was close to shipping, such that only code changes for high severity bugs were accepted.

2. *Nature of the product*

As specified above, Parallel Extensions is a library for supporting multi-threading applications.

Unlike their sequential counterparts, our algorithms might not take the same execution path every time. Hence developing tests that can cover all the possible thread scheduling is difficult. For the hard-to-reach execution paths, we had to use specific concurrency testing tools like CHESS (13).

3. *Stress Conditions*

Some paths could only be reached by long running tests together with the right interleaving. The time taken to execute these scenarios could span as long as a few days. We analyzed these paths and decided that it was not worth the effort to measure the code coverage for them.

Given all these factors, we were able to achieve an effective 99.76% code coverage. We made sure the remaining 0.24% was covered either by using CHESS, writing long running stress tests, or by inspection of the code.

total blocks	**covered blocks**	**dead code blocks**	**missed blocks because of concurrency/stress**
16889	16480 (97.58%)	368 (2.18%)	41 (0.24%)

Table 12: Analysis of uncovered portions of the code in our case

6. Related Work

There have been several studies related to determining how code coverage relates to the strength of the test suite, and the cost of that test suite. Hutchins et al. (1) randomly picked a set of tests and measured their code coverage against the number of seeded faults found. They showed that for test suites of the same size, the test suite with higher code coverage has substantially higher fault detection rates; specifically as coverage goes from 90% to 100%. Hutchins et al. assert that even 100% code coverage does not find all bugs.

Frankl and Iakounenko (4) published similar work, using European Space Agency programs as the software under test. They found an even more stark exponential increase in fault detection as code coverage increases. Andrews et al. (5) used a different approach: first showing that detecting mechanically seeded faults (mutation testing) is a reasonable approximation of detecting real-world bugs, then showing the relationship between code coverage and killed mutants. They found that for difficult to find bugs, the relationship between code coverage and fault detection is exponential, with a sharp increase in the last 10-20%. Andrews et al. also measured the number of test cases needed to achieve coverage, and show an exponential relationship between the number of test cases needed, and the code coverage achieved. They suggest that achieving 100% coverage may be prohibitively expensive.

All three studies, above imply that the last 10% of coverage will find the most bugs, a result that does not match our experience. One explanation could be that in the above lab experiments, the coverage is computed by running randomly selected tests from a known test suite. In the industrial practice of testing, however, test cases are not randomly selected. Usually, a test team will plan-out a set of tests that they believe will find the most bugs. Thus, it is possible that testers develop tests for difficult-to-find bugs first, and test parts that are known to be of high quality less. If this is the case in the industrial context, the cost of going from 90% code coverage to 100% may not be as expensive as Andrews and his colleagues suggest.

Mockus et al. (6) presented two case studies (one each from Microsoft and Avaya) that try to relate code coverage with post-release bugs. They found that high code coverage is associated with low post-release bugs when controlled for pre-release code churn. The authors speculate that testers put more effort into testing components that have a higher propensity to fail, and that higher code churn is indicative of a higher risk of a failure in that component. They also found that classes with higher code coverage had exponentially higher cost. They conclude that getting better than 90% code coverage may not be feasible in most cases. However, we have to consider that if testers tend to cover the most difficult code paths first, the last uncovered sections that are left may be *easier* to test.

7. Conclusions

Code Coverage tools are great when used along with other test activities. Investigations of the code coverage data encourage testers to invest in white box testing which can lead to discovering more scenarios to test. The dead code can be found earlier in the product development cycle and removed promptly in the same release.

Reaching 100% block coverage is without any doubt the best goal to aim for; however with 97% block coverage and the rest 3% reviewed, our confidence in the quality of our product is very high. Even if the eight bugs found represent a small number comparative with the number of bugs found as a result of the other test activities, we found as well a couple of test issues that will improve the testing process going forward. One such issue was the systematic miss of parameter validation for several of our APIs. The other one is the fact that we missed query testing over data sources of specific count, which leads to an entirely different execution path. The dead code found will be removed in the next product development cycle, which will result in increasing the maintainability of the source code.

Despite the current research in the field (1) (4) (5) (6) , our results show that achieving effective 100% code coverage code coverage is affordable. A possible reason may be that testers in general test the complex parts first, and so the remaining areas are not as hard to cover. At the same time, our research showed that the bug value does not increase exponentially as the code coverage increases. However, using the data from the first coverage efforts we can conclude that investigating the code sequences with high complexity could yield a better Return of Test Efforts.

As a result, our recommendation is to prioritize the test work that still needs to be completed before concentrating all the efforts to increase the coverage to 100%. The formulas defined in this paper may be used during the product development cycle to help in estimating how much effort should be put for a targeted code coverage activity. In our case, the customer scenarios are the top priority.

Acknowledgements

We would like to thank our test lead, Shaun Miller, for supporting the project. We would also like to thank Stephen Toub, Boby George and Chris Dern for reviewing drafts of this paper.

References

1. *Experiments of the effectiveness of dataflow - and controlflow-based test adequacy criteria.* **Hutchins, M., Foster, H., Goradia, T., and Ostrand, T.** Sorrento, Italy : IEEE Computer Society Press, Los Alamitos, CA, 1994.

2. *The effect of code coverage on fault detection under different testing profiles.* **Xia Cai, Michael R. Lyu.** 4, s.l. : ACM SIG Software Engineering Notes, 2005, Vol. 30.

3. *Achieving software quality with testing coverage measures.* **Joseph R. Horgan, Saul London. Michael R. Lyu.** 9, s.l. : IEEE Computer Society, 1994, Vol. 27.

4. *Further empirical studies of test effectiveness.* **Frankl, P. G. and Iakounenko, O.** Lake Buena Vista, Florida, United States : SIGSOFT '98/FSE-6. ACM, New York, NY, 1998.

5. *Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria.* **Andrews, J. H., Briand, L. C., Labiche, Y., and Namin, A.** 8, s.l. : IEEE Trans. Softw. Eng., Vol. 32.

6. *Test Coverage and Post-Verification Defects: A Multiple Case Study.* **Mockus, A., Nagappan N., and Dinh-Trong, T.** Orlando, FL : ACM-IEEE Empirical Software Engineering and Measurement Conference (ESEM), 2009.

7. Parallel Extensions to .NET 4. [Online] http://msdn.microsoft.com/en-us/concurrency/default.aspx.

8. Task Parallel Library. [Online] http://msdn.microsoft.com/en-us/library/dd460717(v=VS.100).aspx.

9. Parallel Linq. [Online] http://msdn.microsoft.com/en-us/library/dd460688(v=VS.100).aspx.

10. LINQ to Objects. [Online] http://msdn.microsoft.com/en-us/library/bb397919.aspx.

11. *Data Structures For Parallel Programing.* [Online] http://msdn.microsoft.com/en-us/library/dd460718(v=VS.100).aspx.

12. *A complexity measure.* **McCabe., T. J.** 4, s.l. : IEEE Trans. on Software Engineering, 1976, Vol. 2.

13. CHESS . [Online] http://research.microsoft.com/en-us/projects/chess/.

14. **A. Srivastava, A. Edwards, and H. Vo. Vulcan.** *Binary transformation in a distributed environment.* s.l. : Technical ReportMSR-TR-2001-50, Microsoft Research Technical Report, 2001.

Managing Polarities in Pursuit of Quality

Denise Holmes
Principal, Edge-Leadership Consulting, LLC
503.719.7462 | denise@edge-leadership.com

Abstract

One of the questions for this conference is "can complexity be managed or are we destined for complete chaos?" The key concept in that phrase is that of *managing complexity*, versus being at the whims of a complex system. Sometimes a sense of powerlessness results from treating interdependent factors as independent of one another and then being surprised at the negative impacts that occur because we weren't aware of their interrelatedness. For example, taking the stance "we need to focus on quality no matter what it costs or how long it takes" could result in a wildly over-budget project that is obsolete or of no interest to the customer by the time it is released. This is more likely in a situation in which costs and time to market aren't proactively managed at some level. The opposite also holds true, going for speed at the expense of quality, or cost at the expense of quality might mean losing customers' trust and business, beginning a downward slide that finishes the organization. The reality is that all of these factors are important and influence each other to some degree: quality AND cost AND time to develop. These are just a few dynamics that may be competing with each other in our world.

So, what does this mean for the quality professional of today?

Managing complexity means recognizing differing and critical needs, how to experience the upsides of each need, avoid the downsides, and do it all intentionally, with awareness of the choices being made. The skills to accomplish this, as covered in the paper, include the ability to:

- Recognize system dynamics at play, called polarities, that may be getting in the way of your quality goals;
- Manage these polarities through a deceptively simple process; and
- Apply this process as a communication and conflict resolution tool to create dialogue about what is important and why.

This paper introduces the process of polarity management as a way of seeing, thinking and communicating around opposing dynamics to help you become (or remain) a proactive player in support of your software quality efforts.

Biography

Denise Holmes, Principal of Edge-Leadership Consulting in Portland has a background in designing and delivering experiential leadership development, coaching individuals and teams through change and applying system dynamics to her consulting. Denise built her expert facilitation and coaching skills in the worlds of health care and high tech. Denise holds an M.A. in applied behavioral science from the Leadership Institute of Seattle. She also holds an M.B.A. from Marylhurst University and a B.A. in international studies from the University of Oregon.

1. Introduction

There is no doubt that the world of software engineering is complex in both technical and managerial ways. When overwhelmed by the complexity of a situation we may respond in several ways: freeze and become unable to function; fight and argue with others; or go along to get along while simultaneously believing that whatever we try will never be good enough.

This paper introduces the process of polarity management: a way of seeing, thinking and communicating opposing dynamics so that you can feel more in control within complex systems, increase the choices available to you and those you work with, and find common ground on which to work through conflicts with other stakeholders. Specifically, this paper mixes theory and practical examples to:

1) Define polarities and share examples that apply to software quality;
2) Share the dynamic pattern of polarities: upsides, downsides, and what happens when there's a badly managed polarity;
3) Outline the steps for managing a polarity well over time; and
4) Apply the polarity management process as a communication and conflict resolution tool.

2. What are Polarities?

Polarities are sets of opposing or seemingly contradictory alternatives which over time, can't function well independently. This means that to some degree, the polarities represent constant tensions to manage, not problems with a clear solution or decision point. This definition will become clearer through examples in this paper. The image below represents two opposing poles.

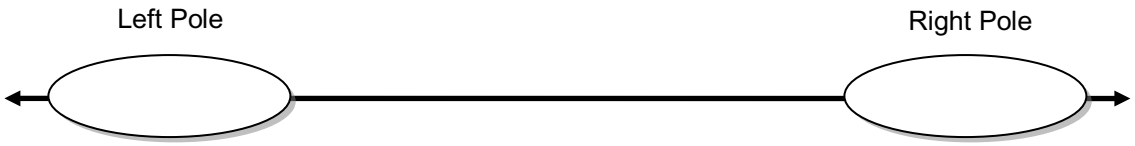


Figure 1: A visual representation of two polarities and the implied tension between them

When a polarity is present, both points of view are accurate, but neither is complete on its own. For success, the polarities are recognized as a "both/and" scenario rather than an "either/or" choice. This is because a focus on one point of view at the expense of the other will result in negative impacts, as will be discussed further below. In the study of polarity management, polarities are also often referred to as opposing dynamics, paradoxes, dilemmas, or competing points of view.

Examples of polarities in the software quality field include:

- *Uniformity and customization:* For your software, what is the right mix of uniformity and customization that is manageable and cost-effective?
- *Stability and change:* How do you balance identifying release criteria with follow-on releases and new features?
- *Advocacy and inquiry:* How might you balance proactively sharing and advising best practices for your product while soliciting and integrating customer feedback?
- *Details and big picture:* What do the coders and testers need to know about the customer's business requirements, both current and future?
- *Task and relationship:* How do team members learn enough about each other to work well together and focus on the work to be accomplished? How do you build trusting relationships with the client so that it supports getting the work done as needed, when needed?

- *Structure and flexibility:* In choosing a software development methodology, how do project teams manage what stays firm and fixed in the processes, and what is malleable, open to emergent factors, and flexible?[1]

A longer list of polarities is listed in the Appendix, though there are infinite combinations of potential polarities.

2.1 Differentiating Between Problems and Polarities

Polarities, or dynamic tensions, are not solvable. There is no magic pill or series of steps that fixes and resolves the tension once and for all so that you can move on. Instead, polarities reflect constant tensions that must be managed, even though the intensity of the tensions will vary over time, based on how well you manage between the polarities.

A situation is more likely a problem to solve rather than a polarity to manage if:

- The situation has an end point and is not ongoing;
- The alternatives are independent, and function well on their own;
- It's an either/or decision-making process (e.g., Do we hire Jane or Marie? Where should we go to lunch? What questions should we include on our survey?); and/or
- The situation is about making choices on a continuum, such as low to high quality. For example, some organizations will make deliberate decisions to allow certain defects that will be fixed with later releases in order to reach early adopters. That said, the problem to solve with this example is identifying the acceptable number of defects. This problem is also an example of a problem nested within one or more polarities, e.g., quality and time to release.

3. The Dynamic Pattern of Polarities

Each polarity represents potential positive and negative results. When mapping a polarity, the positive results are the "upside" and the negative results are the "downside." This reflects where they are placed in the diagram. See the map on the next page for how these results might be represented for the polarity of uniformity and customization in the context of developing software.

[1] Structure and flexibility is a polarity reflected in trends between preferred development tools, for example the Waterfall Model representing a more linear, structured process and Agile representing a more flexible and emergent process.

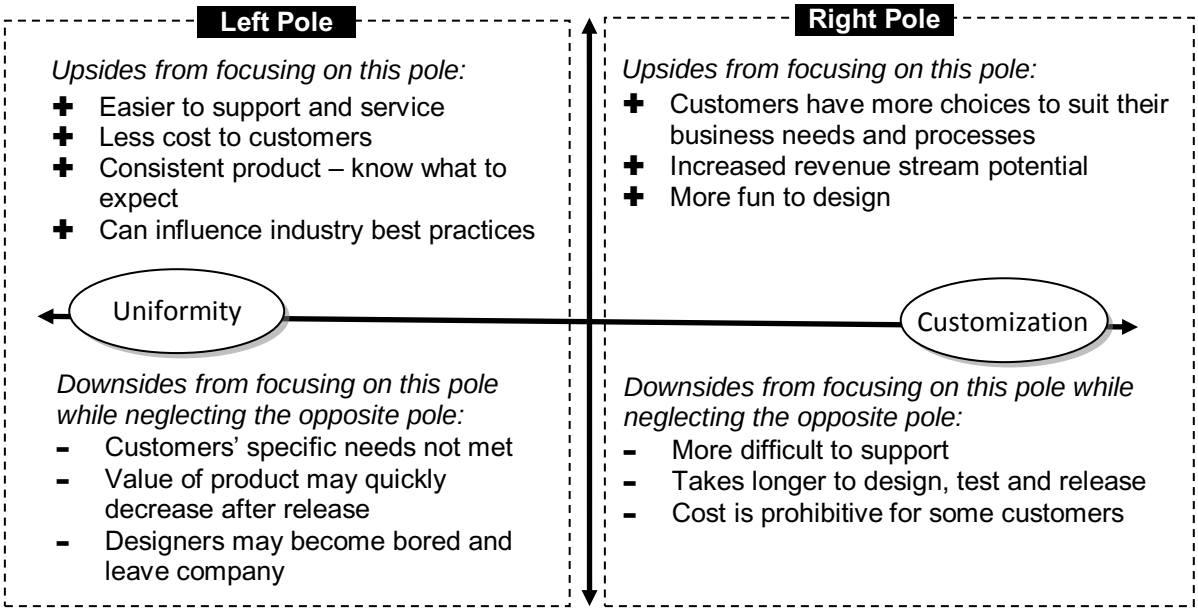


Figure 2: A sample polarity map showing the upsides and downsides of a polarity

The goal in polarity management is to experience the upside of both poles and to minimize the downside of both poles. It does not mean that both poles are equal in priority or level. The degree of uniformity and customization are deliberate choices with the awareness of the costs and benefits of those choices and the interdependent nature of the poles.

We focus on one pole because we want to achieve the attached benefits, e.g. – software that is easy to support and service. When we don't recognize that one factor is part of a polarity, an over-focus on one pole at the exclusion and neglect of the other pole causes us to begin experiencing the negative results associated with the original pole, e.g. – software that won't meet everyone's needs.

Systems thinking literature introduces the concept of *balancing feedback*, where "change in one part of a system results in changes in the rest of the system that restrict, limit or oppose the initial change" (O'Connor & McDermott, 1997). Barry Johnson has been working on the Polarity Management Model and set of principles since 1975. He created a way to map the system oscillation that occurs between the upsides and downsides of a polarity.

3.1 A Poorly Managed Polarity

Let's take the opposing tensions of *time to release* and *quality* and imagine a poorly managed polarity over time:

1) An organization has prided itself on beating competitors to market and having early adopters of its product. The rewards have included premium prices for new software releases and a highly motivated excited workforce that prides itself on its creativity and competitiveness. However, parts of the organization have little patience for quality testing and protocols.

2) The organization realizes that it has been receiving too many customer complaints about defects in the software. Customers are switching to other, more stable products offered by competitors, and bad reviews about quality have caused a significant decrease in new customers. The company has also had to pay significant amounts of money to clients who experienced catastrophic failures and threatened to sue over damages. Executives in the organization recognize that something needs to change and make the claim "quality is now our number one priority."

3) Strict quality protocols are put into place along with a highly motivated, newly created quality assurance team that is determined to not approve releases with defects. New marketing campaigns are created around a commitment to quality assurance and customers are becoming reassured by promises of worry-free implementations. However, the organization is incredibly cautious and has completely prioritized quality assurance at the expense of any pressures to release by a promised timeline.

4) Customers start to complain about continued delays in implementation and start cancelling orders as the organization tries for a perfect product. Testers become nervous about whether they've thought of every possible testing script, even though the Business Analysts had provided information and helped write testing scripts for the needs identified by the customers. Multiple departments in the organization become frustrated at the Quality Assurance Department and begin pressuring the Executives to focus again on developing "good enough" software products quicker. Executives agree and start minimizing the role of the Quality Assurance Department in the development and release cycle.

5) Phases 1 through 4 repeat over time, unless recognized and proactively managed or until the organization goes out of business.

3.2 How this Maps

The black arrow in each diagram below shows the dynamic shift within and between poles.

1. The organization focuses on the actual and perceived advantages of one pole (time to release) without also prioritizing the other pole (quality).

   This causes the organization to experience the downsides of that pole.

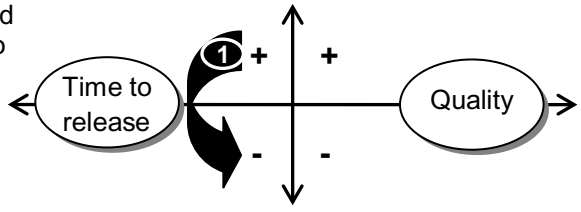


Figure 3: Dynamic showing shift from upside to downside of the left pole.

2. Experiencing the downside of one pole (time to release), causes the organization to react so that it can prioritize and experience the advantages of focusing on the opposite pole (quality).

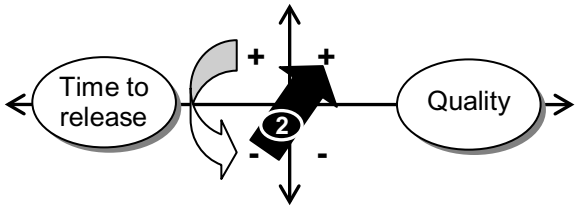


Figure 4: Dynamic of self-correction, showing the shift from the downside of one pole to the upside of the opposite pole.

3. The focus on the actual and perceived advantages of one pole (quality) without also prioritizing the other pole (time to release) eventually causes the organization to experience that pole's downsides.

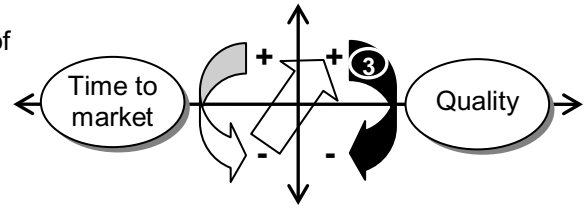


Figure 5: Dynamic due to over focusing on the right pole at the expense of the left pole.

4. Experiencing the downsides of this pole (quality) causes the system to adjust again so that it can prioritize and experience the advantages of focusing on the opposite pole (time to release).

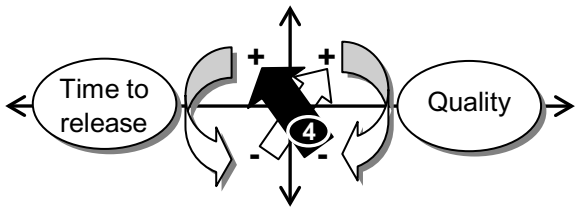


Figure 6: Dynamic of self-correction, recognizing the downsides of one pole influencing a focus on the opposite pole in reaction – back to the starting place.

Over time, if the system continues to treat these opposing tensions as an "either/or" choice, it creates an infinity loop: The paradox of polarities is that "in order to gain and maintain the benefits of one pole, you must also pursue the benefits of the other" (Johnson, 1992).

Figure 7: The system oscillation that occurs between polarities

3.3 Recognizing the Polarities in Your System

You might be thinking that the above example is extreme and that your organization has already put steps in place to balance the focus on time to development and quality or the other polarities listed above. To recognize relevant polarities in your system, ask yourself and others:

a) What ongoing issues or conflicts are you regularly facing and trying to solve?
b) Where are you feeling stuck?
c) What are you or those around you trying to move from?
d) What are you or those around you trying to move towards?

Each of these questions will give you clues to polarities relevant to you.

4. Managing Polarities

Awareness is the first step to proactively managing polarities so that you can experience the benefits of both sides and have early warning signs that allow you and others to self-correct before negative ramifications become too serious.

The steps for managing a polarity are:

1) Identify the polarity you want to manage better.

2) Map the polarity to form a complete picture.
   a. Use value-neutral words for the poles (e.g., flexibility and structure vs. willy-nilly and rigid)
   b. Identify the critical upsides and downsides of each pole. The upsides are what you value and want to maintain, while the downsides are what you fear and want to reduce, avoid or eliminate.

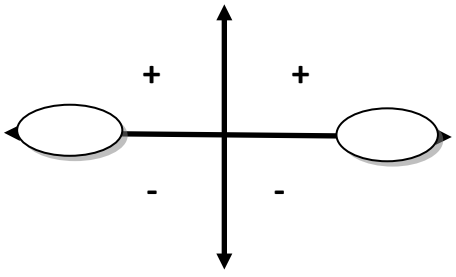


Figure 8: Blank polarity map

   c. Identify where you are right now, based on the behaviors or impacts that are showing up. In what ways are you out of balance? Where are you experiencing the upsides of either pole and where are you experiencing the downsides?

3) Identify action items that will help you leverage and experience the upsides of both poles. The action items can address both poles at the same time, or be individual actions that support each pole separately, as long as both poles are addressed. When action items are chosen, assign accountability – who will accomplish each action item by what date. Have each action item be specific, measurable, achievable, reasonable, and time-oriented (SMART).

4) Identify critical warning signs that will alert you or the appropriate persons when the system begins sliding into one of the downsides. Like with the action items, specify how the warning sign will become known and communicated. For example, would the warning be customer complaints and/or project delays? If so, how many or about what types of things? Who (individual or group) would first notice the warning sign and how would that be communicated to key stakeholders? What reports, charts or dashboards does your organization use that might signal warning signs?

5. Using Polarities as a Communication and Conflict Resolution Tool

Any time there are individuals and groups who have competing priorities and points of view, conflict can easily arise and it becomes an issue of who is right or who has more power. The polarity management process can help you achieve shared understanding and resolve conflicts, because it:

1) Explains polarized viewpoints,
2) Clarifies what the different perspectives value and fear about their point of view, and
3) Helps identify the common purpose and common fears that can bring both perspectives together.

In the Dalai Lama's Book of Wisdom, he wrote:

> If we look at the situation from various angles, such as the complexity and inter-connectedness of the nature of modern existence, then we will gradually notice a change in our outlook, so that when we say "others" and when we think of others, we will no longer dismiss them as something that is irrelevant to us. We will no longer dismiss them as something irrelevant to us. We will no longer feel indifferent (2000).

His words describe the outcome of using the polarity management process between divergent points of view: an appreciation of the other side and recognition of how it may interconnect in meaningful ways with your perspective.

5.1 Polarized Points of View

In conflict situations, each pole typically represents one side's "Truth" and the other side's "Error." When one side is perceived as right and the other wrong, it represents either/or thinking. In these cases the side with the most power or the side which is currently most entrenched wins until the pain of the current situation becomes significant enough to overcome the status quo. Do any of the following statements sound familiar?

- "Software should be released 100% free of defects!"
- "Testing takes too much time and doesn't take into account all the real business scenarios used by the customer."
- "The people in charge don't appreciate us and how important it is to complete thorough testing protocols."

These types of statements are indicative of polarized views.

5.2 Values and Fears

Once you've identified the polarized points of view, list what each side values and fears. What you value about your perspective shows up on the upside of your pole and what you fear is on its diagonal downside. And the same is true of the other side's perspective. For example, if you are wrestling with the polarity of planning and flexibility, and you want more time and value placed on planning, your view might be reflected as:

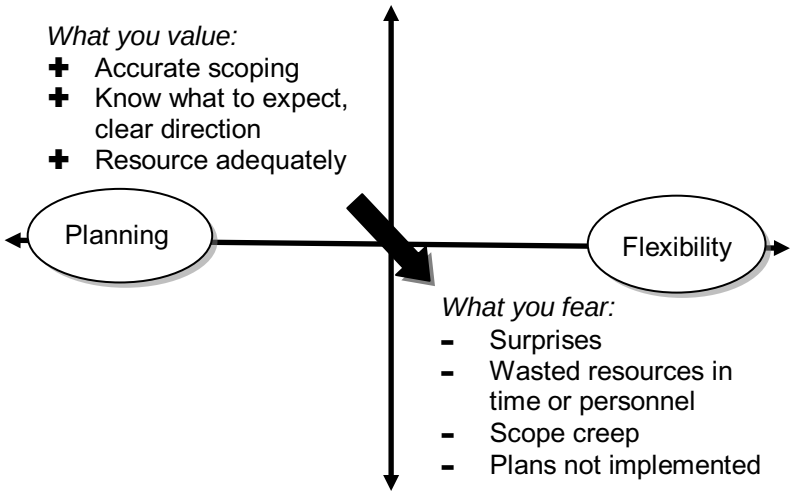


Figure 9: Values and fears mapped from the planning perspective

The other person's view might be reflected as:

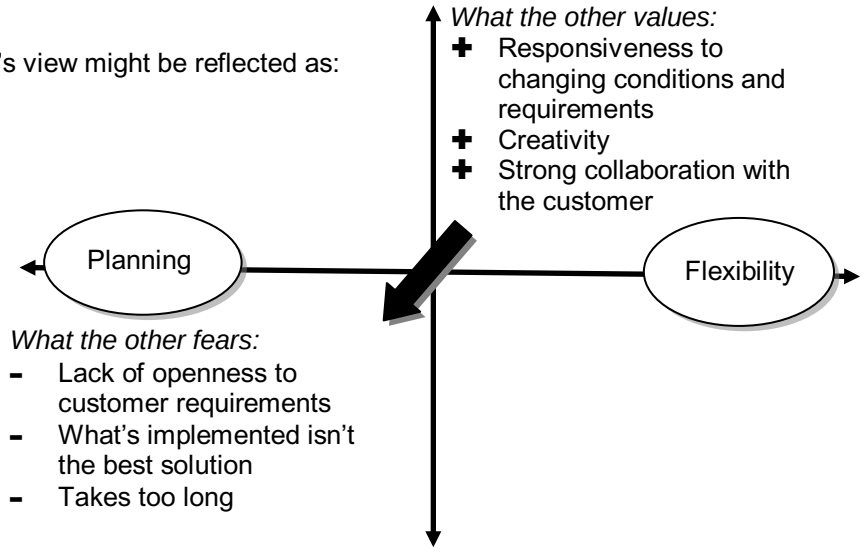


Figure 10: Values and fears mapped from the flexibility perspective

When you are in a polarized situation, each party is already anticipating the downsides of the other view – either because they experience those downsides, have experienced them in the past, or are simply afraid that they might occur. Recognizing this relationship between the two sides is critical for change management efforts as well, in creating a fuller picture of the situation, legitimizing multiple perspectives and anticipating potential resistance.

5.3 Higher Purpose and Deeper Fear

The unifying step within this process is to identify the higher purpose and the deeper fear that is common to both poles. Examples of potential higher purposes for two sides in conflict over planning and flexibility may be anything from desire for a working, viable product that fully meets or exceeds customer expectations to continued success and profitability of the organization. A shared deeper fear may be no faith in the software eventually resulting in no faith or confidence in the organization.

The higher purpose goes above the upsides and the deeper fear goes below the downsides, as shown in the combined example of the polarity map on the next page.

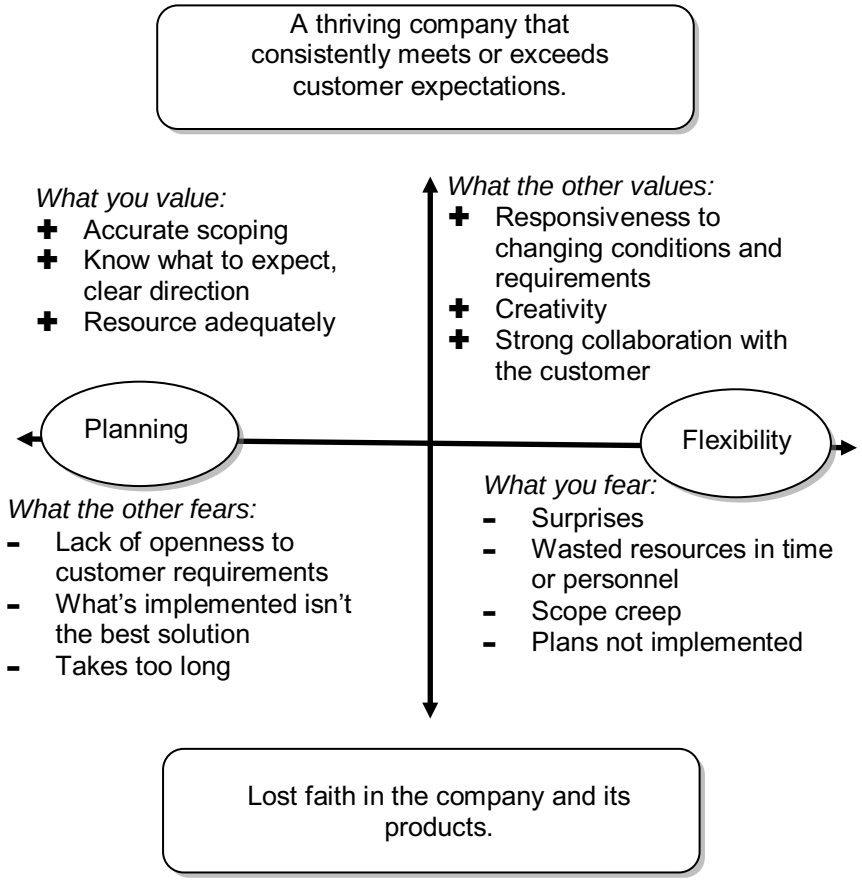


Figure 11: A complete polarity map, applied to a conflict situation

5.4 Using this for Communication and Conflict Resolution

So, how might you apply this process?

1) Use it for yourself, to gain greater awareness of what other perspectives might be, and to understand why those beliefs and opinions are valid. Change how you interact with others by not labeling their perspective as misguided (or worse). Mention how you can understand how they might be worried that "x" could result in "y". Collaborate with the other person(s) on how to minimize the risks of the downsides occurring.

2) Use it as a sales tool, such as with executives or customers. Mapping the different perspectives will help you make a persuasive case for change. You can clearly articulate what you see as the shared purpose, the values and advantages to both perspectives being shared, and what you plan to avoid (e.g., the downsides). You can either have potential solutions already in mind, or collaborate with the applicable stakeholders on appropriate actions.

3) Use it within a team or between teams by naming one or more polarities, collaborating on brainstorming and decision-making to fill in the map, sharing perspectives, identifying action items and warning signs, and assigning accountability as applicable.

6. Multarities

The scope of this paper is to address a process for mapping and managing two polarities at a time. In reality, we are often dealing with "multarities," multiple opposites that are interacting with each other at any particular time. For example, most projects must balance the "Iron Triangle" (Ambler, 2003-2006) of cost AND time AND quality AND scope. Those four factors actually reflect six polarities:

1) Cost and time
2) Cost and scope
3) Time and quality
4) Cost and quality
5) Time and scope
6) Quality and scope

You may also find that one polarity has several sub-polarities nested within it. In choosing what to focus on, identify what is most currently an area of pain or discord for you and applicable stakeholders.

7. Conclusion

This paper presented a way of seeing and managing complexity, specifically how to identify and manage opposing dynamics, or polarities. This is not a process applicable for every situation, such as when addressing a problem vs. a polarity. Additionally, polarities are everywhere, but not every polarity needs your attention. Use this process to work the polarities that need more active management to help you achieve your goals.

In summary, the ability to manage polarities will help you:

- ☑ Recognize the oscillation that occurs in any system or organization when polarities aren't managed well. Awareness of this dynamic will help you feel less powerless in the midst of complexity and help you proactively identify choices to move closer to your ultimate goals and higher purpose;
- ☑ Get unstuck when you feel caught between conflicting choices or priorities;
- ☑ Recognize when a recurring problem you keep solving is really a polarity to manage;
- ☑ Increase your understanding of what causes resistance and how to approach communication and change efforts in more open and engaging ways; and
- ☑ Create dialogue with others about what is important and why, in order to reduce conflict and resistance and create momentum towards a shared goal.

Have fun with this process as you use it to help manage the complexity around you. Especially, use polarity management to stop repeating the same steps while expecting different results and to focus on important, shared goals between stakeholders instead of getting stuck in defending a specific position.

Appendix

Acknowledgements

This author would like to thank her official PNSQC reviewers for their thoughtful feedback and insights: Ian Savage and Douglas Reynolds. Your time and expertise in support of this paper are valued!

References and Additional Resources

Ambler, S. (2003-2006). *The "broken iron triangle" software development pattern.* Retrieved from http://www.ambysoft.com/essays/brokenTriangle.html

Boehm, B. and Turner, R. (2003). *Balancing agility and discipline: A guide for the perplexed.* Boston: Addison-Wesley.

Christensen, K. (2009, Winter). Building shared understanding of wicked problems [Electronic version]. *Rotman Magazine*, 16-20.

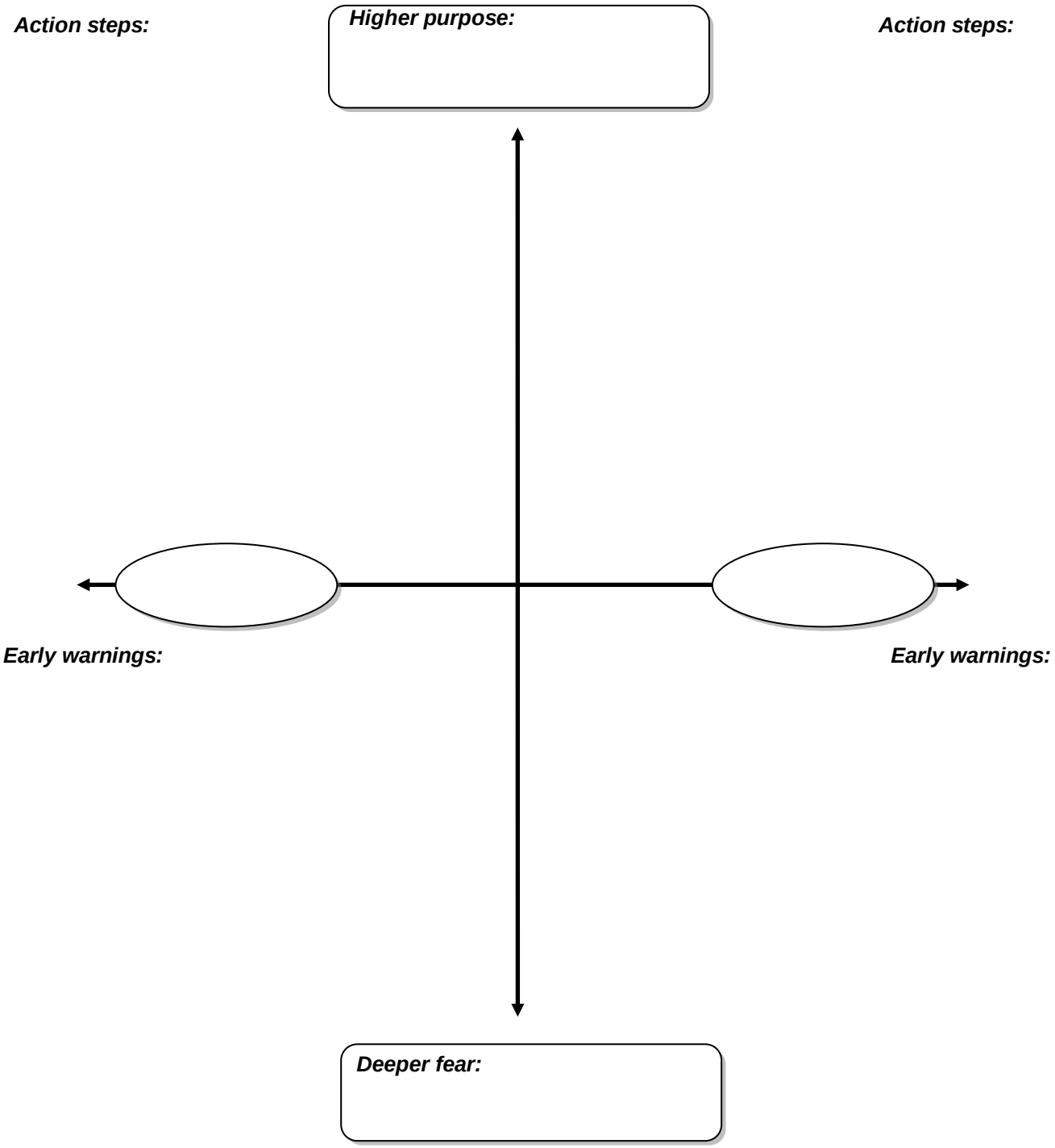
DeMarco, T. and Lister, T. (2003). *Waltzing with bears.* New York: Dorset House Publishing.

Fletcher, J. and Olwyler, K. (1997). *Paradoxical thinking: How to profit from your contradictions.* San Francisco: Berret-Koehler Publishers.

Johnson, B. (1992). *Polarity management: Identifying and managing unsolvable problems.* Amherst, MA: HRD Press.

O'Connor, J. and McDermott, I. (1997). *The art of systems thinking: Essential skills for creativity and problem solving.* San Francisco: Thorsons.

A Blank Map


Action steps:
Higher purpose:
Action steps:
Early warnings:
Early warnings:
Deeper fear:

Lean System Integration at Hewlett-Packard

Kathy Iberle
kiberle@kiberle.com

Abstract

Discover how HP is applying Lean principles to drive the integration of large systems, resulting in both higher quality and higher productivity.

In the HP printer business, Lean integration is

- making complex programs easier to manage by providing visibility into what the product can and cannot do at any point in the development
- improving the customer's experience by making customer workflows functional and visible early and often throughout the lifecycle
- reducing cost by driving synchronization of delivery across technology components

This paper provides an introduction to the methods known as Lean Software and Systems Development. Lean is able to handle situations which are difficult to handle using the most commonly known agile methods, such as large, complex, and partially waterfall systems, by applying methods deriving from queuing theory and statistics.

Lean methods are demonstrated in this paper with examples and results from actual projects in the HP Inkjet and LaserJet businesses.

Biography

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.

Acknowledgements:
A big thank you to Dr. Laurian Dinca and to Chris Moehring, for providing the data and project history which was included in this paper.

First published at the Pacific Northwest Software Quality Conference 2010

Introduction

What is Lean Product Development?

Lean Product Development is a paradigm for developing products which views the development organization as a system or machine to produce profit. Lean Product Development uses proven engineering, mathematical, and statistical methods to maximize the system's output of profit by engineering a rapid, smooth flow of saleable features and capabilities through the development organization. This is done by

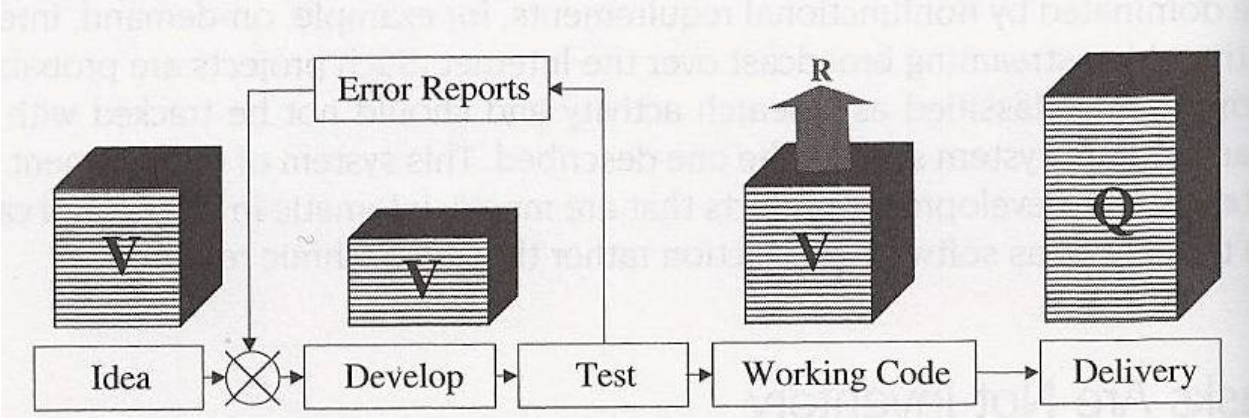
1. Modeling the work of the organization as a system to turn ideas into money. The system model consists of activities and wait states or *queues*.
2. Observing that unfinished work isn't free - there is money tied up in it.
3. Applying a body of mathematical knowledge known as *queuing theory* to optimize the profitability of the system. This optimization requires
    - minimizing the money tied up in unfinished work
    - minimizing the operating expense
    - maximizing the output of saleable product

Let's look at these ideas one at a time.

First, what does the work look like when modeled as a system for turning ideas into money?

Lean Product Development models the development system as items moving through a series of activities and wait states. Each activity takes more than zero time. Whenever items have to wait to be processed through an activity, a line or *queue* forms, where items wait to be served.

Here's a very simple model of software development, showing four activities (the boxes) and four queues (the stacks waiting to be processed through the activity).



(Anderson 2004, p. 53)

In this model, we can measure several things

- *Throughput* = stuff of value to the organization. "Value" means it is ready to be sold, or used to reduce internal costs, or otherwise immediately provide concrete value.
- *Investment* = money invested in material waiting to start, in process through the system, or waiting to exit from the system.
- *Operating expens*e = money spent to run the system

In order to measure anything at all, there must be discrete chunks moving through this system. In a traditional waterfall development model, the entire body of work moves through the system as a single large chunk. In Lean Product Development, the work is broken into smaller chunks which move through the system sequentially, one after another.

A key difference between Lean Product Development and waterfall is that Lean Product Development insists that the chunks moving through the development system ***must*** be saleable features, ***not*** tasks. This ensures that all measurements and analysis of the system will focus on maximizing the throughput of *saleable features and capabilities* (and thus profit) rather than the throughput of *tasks*.[1]

This will sound very familiar to anyone who has worked with agile development. I'll explain the difference between agile development and Lean Product Development in a bit.

Second, why does unfinished work cost money?

The cost of unfinished work is easier to see in an example using physical objects. Let's look at a manufacturing line.

On a manufacturing line, the system consists of a number of assembly steps, each performed at a different assembly station. The partially completed items move from one assembly station to the next. At each station, some parts are added to the partially-assembled item.

In the old days, it was assumed that maximum output was achieved by having every assembly station utilized at 100% capacity - that is, running full tilt all the time. This required having enough parts on hand to keep every station running regardless of what was happening at other assembly stations.

This makes sense, except for one thing: the different assembly stations *don't* run at the same speed all the time. The variation between stations has two results:

- Each assembly station needs a stockpile of parts large enough to handle the fastest possible input.
- Partially completed items pile up in a queue in front of the slower stations.

The stockpile of parts and the partially completed items both contain parts not yet part of a saleable product. This is the *inventory*. The inventory ties up money in several ways.

- Money invested in purchasing the parts
- Owning and maintaining shelving & storage space for parts
- Maintaining an elaborate tracking system and related processes to keep track of the multitudes of parts
- Time spent by people to communicate with each other and the tracking system concerning the status and location of the parts

Just-in-time (JIT) manufacturing practices were introduced in the 1980s to minimize this investment. JIT focuses on finishing items from start to end as rapidly as possible rather than utilizing every assembly station 100%. Basically, the whole system runs at the speed of the slowest assembly station. Parts are pulled into a workstation just before they are needed, rather than stockpiling parts. This means there are smaller stockpiles and fewer partial assemblies sitting in queues. Using JIT, it was often possible to reduce the number of parts on hand at any given time by 80% or more. [Reinertsen 1997]

Because there are fewer parts sitting around, the shelf space needed was vastly reduced. The tracking systems became much simpler, sometimes just a set of index cards instead of a ledger-based or electronic system. The total amount of communication needed was reduced. All this simplification not only made life easier, it also freed up huge amounts of cash which could be invested in other parts of the business.

In software development, we also have an inventory of material which is not yet saleable. The agile community has recognized from its start that this inventory likewise ties up money and resources. Software development's inventory consists of ideas, specifications, design documents, code fragments,

[1] In the case of development for in-house use, "saleable features" are replaced by "features which reduce labor or cost within the business". The focus is always on providing usable value.

untested code, defect reports, etc. Just like parts in a manufacturing process, the documents and code produce no profit until the final product is finished and shipped. The software inventory is not as obvious because it physically exists as bits on a drive rather than parts on a shelf, but the costs are much the same:

- The labor invested in developing the ideas, documents, and code
- Owning and maintaining server space for storage
- Maintaining configuration management, requirements management, and defect management databases to track the location and status of the multitudes of pieces.

In addition, unlike the physical parts in a manufacturing process, the half-finished software inventory usually loses value over time. The environment changes, the customers' needs change, or the developers forget the details & have to reinvestigate. If a long enough time passes, the whole thing has to be thrown away and done over.

Third, how can queuing theory help us optimize our development system?

There is a substantial body of mathematics known as *queuing theory* which accurately predicts the behavior of a system of activities & queues under various conditions – high-capacity activities, low-capacity activities, big batches of items, small batches of items, variability in the "size" of the items (the amount of resources required to process the item), and so forth. Once we've split our work into chunks, and modeled our system as activities & queues, we can apply queuing theory to speed up the throughput of the chunks through the system.

A set of chunks of work will usually move through a system fastest when the chunks are started off one at a time (or a few at a time), rather than all at once. The number of chunks in the system at any given time is known as the *work in process* or WIP. Keeping WIP low reduces the amount of tracking and communication needed, just as a smaller number of parts in inventory reduces the amount of tracking needed.

Low WIP has one other very helpful effect – it reduces the amount of task-switching. A human being cannot switch from one task or topic to another for free. The simplest switches take a few seconds. In a switch between complex tasks, the worker may take as much as ten minutes to come fully up to speed on the second task. The time spent task-switching is mostly wasted time. When we have multiple projects in process at the same time across multiple people, the cumulative cost of task-switching and the additional communication needed is huge. Cutting down the WIP is like getting free money, or more accurately, free hours of work.

The traditional waterfall process of setting all features into motion in lockstep produces the highest possible WIP for the workers. The WIP is much lower when a set of features is designed, coded, tested, and fixed to completion before starting the next set.

Agile and Lean Product Development Compared

Agile development also recognizes the value of splitting work into small, saleable chunks, and minimizing the amount of unfinished work in progress at any given time, and minimizing task-switching. The main difference between agile and Lean Product Development is that most of the agile community explains why agile works in terms of heuristics or rules-of-thumb such as "the last responsible moment", whereas Lean Product Development explains why Lean works in terms of a system model and queuing theory. Lean Product Development presents the queuing theory algorithm and shows the user how to work out the answer for a type of situation when it appears in their particular system.

Most agile development principles or heuristics are the answer derived when queuing theory is applied to a small, relatively simple development system. They are correct, for that particular type of system. However, when the system gets larger and more complex, application of queuing theory often leads to a different answer. This allows Lean Product Development to address large, complex development

systems clearly and systematically, whereas the agile methods often struggle with these systems. It also allows Lean Product Development to be used in mixed systems, where part of the system is still waterfall.

A good example of this is the practice of Continuous Integration. Agile and Lean both agree that a short time between making an error and receiving feedback on the error will reduce the time needed to diagnose and solve the error. A developer will more quickly find and fix a problem in code he wrote a few days ago as opposed to code written a few weeks or a few months ago, because there is less time spent remembering or re-learning the code.

Since a short feedback loop saves time, should we make the feedback loop as short as we possibly can? In *Extreme Programming Installed* [Jeffries et. al. 2001, p. 78], the authors say "The extreme solution, of course, is to integrate as often as possible. We call it *continuous integration*. A good XP team will integrate and test the entire system many times per day. Yes, many times per day." The authors follow this with an observation that this is impractical in many development systems because the build takes too long, and exhort the reader to reduce build time to near zero. Continuous Integration is a heuristic.

In contrast, Lean looks at the economics of the entire system, comparing cost and value. Agile and Lean both agree that earlier feedback adds value. Lean asks more questions: How much value is added? How does that value vary over time? What does it cost to add that value?

Feedback on errors has an inherent value to the developer – I need to know about the error. There's also some additional value to me if I find out about the error before I forget the context, because I don't have to re-learn the context and this saves me some time. In my experience as a developer, this additional value of fast feedback also changed depending on the type of error reporting. The value of feedback on coding errors dropped off quickly over a matter of hours, reaching zero about three days after I wrote the code. After that, it didn't much matter whether I got the feedback on day 4 or day 24 – I'd forgotten the exact logic. This is why compiling and unit testing immediately is so important. The additional value of a design error dropped off more slowly, such that feedback within three or four days was nearly as good as feedback on the first day after writing the code.

Now let's look at the cost of getting this quick feedback. Earlier feedback is usually achieved by getting feedback more frequently. This means the developer has to submit code into integration more frequently. Depending on what the integration & subsequent feedback consists of, this submission may not be zero cost.

For instance, suppose the developer is required to run a set of check-in tests before checking in the code. Routinely getting feedback on day N+1 requires checking in every day, which in turn means running those check-in tests every day. This cost is not zero. So, the additional value to the developer of quick feedback is offset by the cost of the check-in itself. At some point, the check-in becomes more expensive than the value of the feedback. After that making the integrations closer together doesn't improve the overall system – it makes the overall system worse.

The exact point at which this happens will be different for different organizations, depending on the cost of the check-in, and the additional value of the quick feedback. The Lean approach to this problem is to
1) Find the approximate optimal point and integrate that often – but not more often
2) If the length of the feedback loop is still a significant problem for the overall system, work to reduce it.
As the Poppendiecks say [Poppendieck 2007, p. 202], "attack the set-up time and drive it down to the point where continuous integration is fast and painless."

The tools of Lean Development allow you to look at a system where the cost of checking-in isn't zero, or the feedback is time-consuming to interpret, or any number of other non-ideal situations, and figure out where the sweet spot is – the point at which overall development cost is lowest. These are very powerful tools.

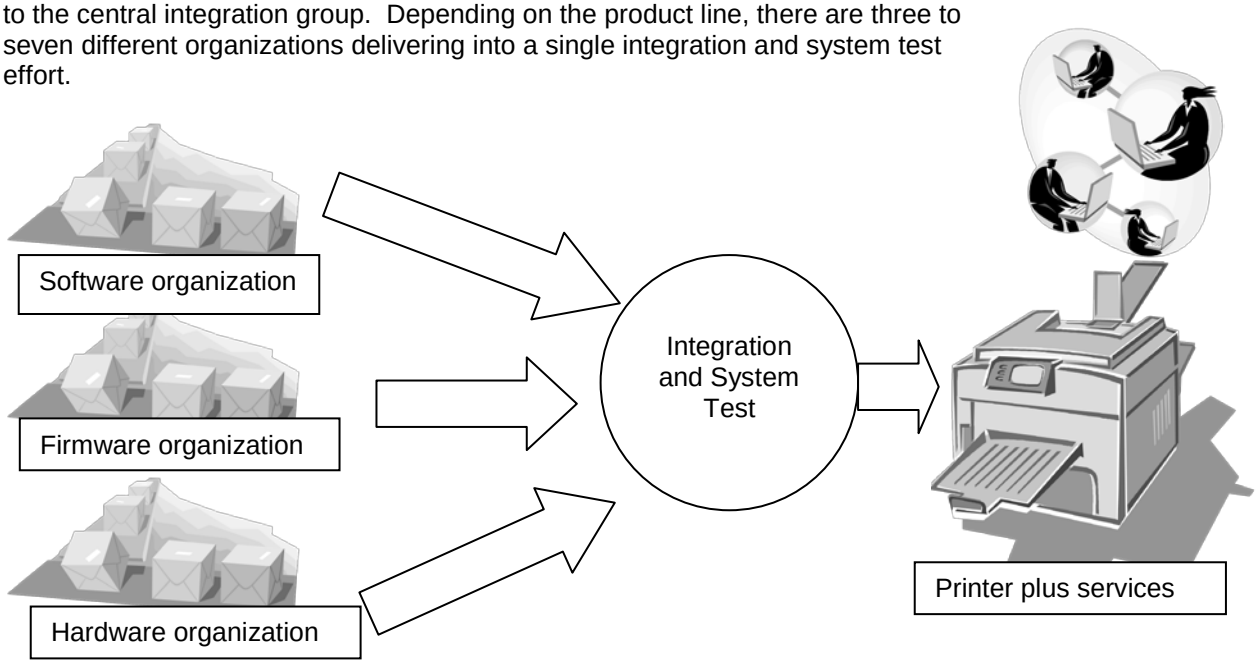
Lean Development Applied – Product Integration at HP

Let's look at a real-life application of Lean Product Development to a system which is large, complex, and in some ways downright messy. This is the story of our organization's application of Lean Product Development, and the benefits we have derived to date.

Starting Point – Agile Development of Subsystems, Waterfall Development of Systems

Our organization was formed in early 2009 as an integration and test group serving both Inkjets and LaserJets. Hewlett-Packard (HP) releases a large number of Inkjet and 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 teams are organized into several large groups, each of which delivers a major subsystem which will be incorporated into multiple products.[2] Most of the groups are now using some form of agile or incremental delivery.

The end result of this is much like a set of conveyor belts, where each organization is delivering features to the central integration group. Depending on the product line, there are three to seven different organizations delivering into a single integration and system test effort.



Despite the fact that each of the subsystem organizations is independently using some form of agile development, the product level development still felt very "waterfall":

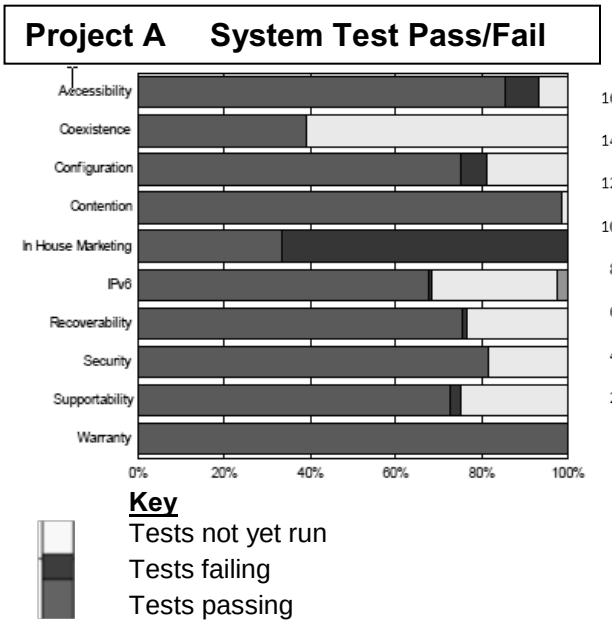
- System test started quite late. Most features were not testable earlier than "code complete" because one organization or another had not yet submitted their piece of the feature.
- Subsystem organizations would receive defects from system test on some code well after the code had been "finished" from the subsystem's perspective
- Final release schedule was rather unpredictable. The progress of individual organizations did not accurately predict the progress of the overall program.
- There was also a hidden cost – the cost of fixing defects was higher than strictly necessary, due to the length of time between releasing code to integration, and the eventual receipt of defect reports from system testing.

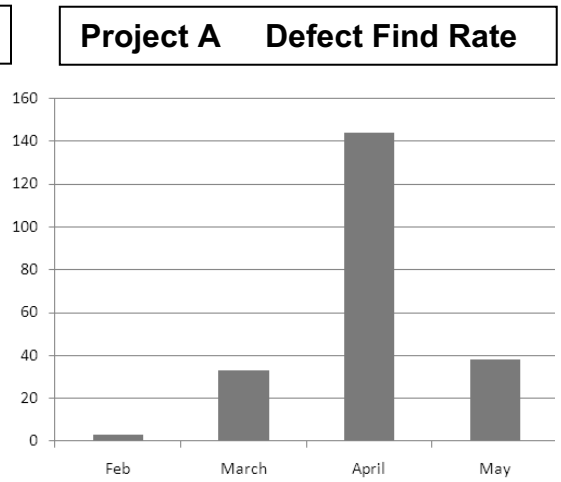
[2] Developing shared software which is configured for individual products is known as *Software Product Lines* or *Software Product Family* style of development. This is a large and interesting topic, but outside the scope of this paper. See the reference list for references, or Google "software product lines".

Let's look at Project A, a typical high-end LaserJet project prior to the introduction of Lean Product Development.

System testing consisted of running a large number of tests which examined end-to-end functionality of the system. The tests were organized by capabilities or quality attributes, such as security, compatibility, recoverability, coexistence, and so forth. The organization of the tests and the test execution plan both assumed that all the subsystems would all be delivered to system test at once, with all features functional. There was no easy way to identify tests for given features and run those earlier or later than the whole.

Typical reporting from system test included various representations of the defect find rate (shown to the right) and snapshots of system test pass/fail status (shown to the left).





System test ran effectively for only three months of a much longer project (typically twelve months or so), and the bulk of defects found by system test were found in a single month near the end of the project. This created an extremely long feedback loop for code written near the start of the project.

The traditional reporting for system test measured progress at passing a standard set of capability-based tests. This didn't tell the Marketing department anything about the status of specific new features.

The newly formed system test and integration organization wanted to start system test earlier and reduce the length of the feedback loop. However, other groups had tried this in the past with limited success. A new approach was needed. The system test organization decided to attempt Lean Product Development at the system level.

Let's look at how we did this and what happened.

Breaking the Work into Chunks – the Slivers

The first step in applying Lean Development principles is to break the work into **chunks with business value**.

Customers don't purchase tests. They purchase features and capabilities which can do work for them. The first big paradigm shift for this group was moving from focusing entirely on *tests* to focusing on *features and capabilities*. This meant we needed to re-organize our tests around the features and abilities in which the user was interested. The tests were already organized by capability, but the feature aspect was missing.

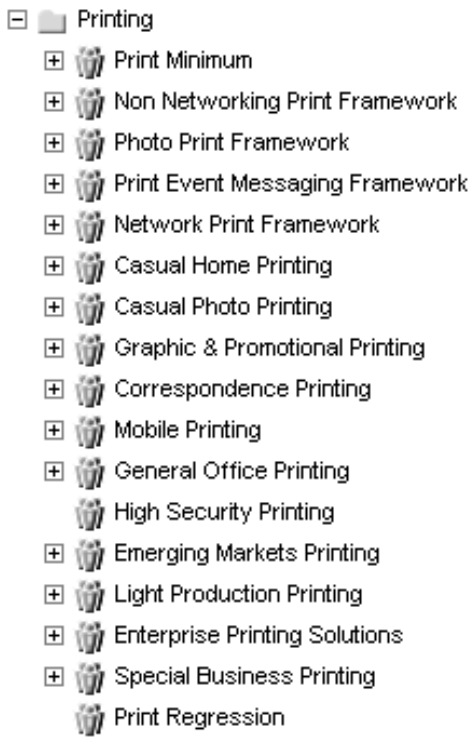
Essentially, we were looking for requirements. Most of the subsystem organizations were using "user stories" and/or lists of functional requirements, either in spreadsheets or in databases. We first looked at those data stores to see if we could identify useful "chunks" at a system level. "Useful" meant:

- saleable (have clear and understandable value to a user)
- applicable at a system level – not specific to one subsystem
- broad enough to avoid drowning us in millions of chunks & the accompanying overhead

We found that, between the multiplicity of technologies in use, and the lack of a common method for identifying and storing *system-level* requirements as opposed to subsystem requirements, we could not easily identify a single consistent set of system-level requirements. We also observed that well-written system-level requirements could be generic enough to be applicable to multiple related products, especially when a user scenario was involved.

We concluded that we could best drive our test re-organization by writing our own high-level requirements, leveraging off of existing system-level requirements whenever possible. We decided to basically take the idea of a user story up one level, and define the "chunks" as a set of user scenarios focused on a particular type of customer doing a particular type of work. The chunks are known as *slivers.*

A typical set of slivers is shown to the right, for the feature area "Printing".



The first few slivers are "getting-started" slivers:

- Minimum entry criteria for this feature area: for instance, can the system print at all?
- Integration slivers (usually named "framework"): do the various subsystems talk to each other successfully? These are often gray-box tests.

The rest of the slivers in this example are user scenario slivers, as suggested by their names. Those usually comprise the bulk of the slivers.

Each sliver contains in its definition:

- A description of the customer
- Specific things the customer wants to do
- How we would know if the customer was satisfied

Here's the definition in a typical User Scenario sliver.

> **Title: Casual Photo Printing**
> **Customer Description:**
> A user who casually prints photo content in a home or office using common photo printing applications. Typical usage is on standard and custom photo types and sizes. The number of users in this sliver can range from one to five. These users can exist on mixed OS platforms (i.e.; Mac, Windows) and varied connectivities (i.e.; USB, Wired, Wireless). These users typically exist in an unmanaged environment.
>
> **As a User I would like to print:**
> 1. 4x6, 5x7, and 8x10 photo prints
> 2. Day-to-day family/ friends pictures
> 3. Scrapbooking
> 4. Facebook - Social networking sites
> 5. Snapfish - Online picture storage
>
> **Customer Satisfier:**
> 1. My photo jobs print right the first time!
> 2. Image quality looks comparable to what user sees on pc.
> 3. Image quality is consistent over time.
> 4. Draft quality provides useful output.

You may notice that the sliver isn't very specific about the product under test. That's because the slivers are designed to be reusable across related products. The integration and system test group typically has around 20 different devices in test at any given time, most of which share large sets of features with each other. In addition, there are many user scenarios which are common to large groups of printers, particularly within individual product lines. We decided to take advantage of the enormous overlap between individual printers by defining reusable slivers whenever possible.

Once the slivers were defined, we had chunks of a reasonable size with clear user value. Next, we needed the ability to measure completion of the chunks, which means we needed completion criteria.

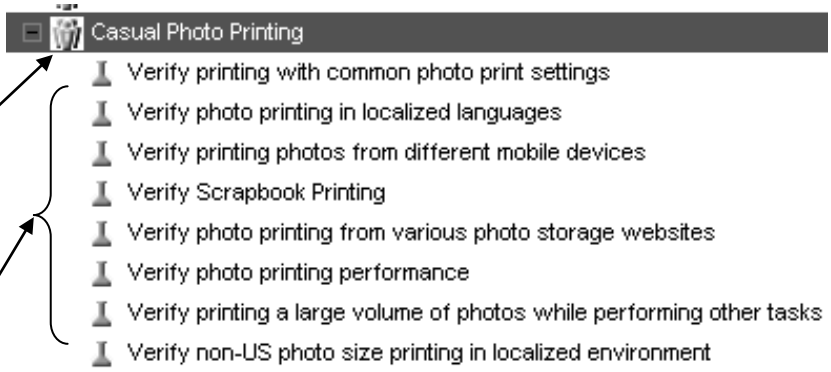
The completion of our slivers is defined as "passing system test"

- A sliver cannot enter system test until all involved subsystems have delivered a version of their subsystem which supports this sliver.
- A sliver cannot enter system test until all subsystems are passing earlier levels of testing at subsystem levels.
- System test starts with a simple entrance test to verify that the subsystems have been successfully integrated and are at a reasonable level of quality.
- The rest of system testing consists of running the tests meant to check the behavior specified in the slivers.

Since the completion criteria included running tests, we needed tests!

This is where it becomes very useful to have an integrated requirements-test management system, such as HP's QualityCenter. We wrote our sliver definitions and our test design into the requirements module of QualityCenter.

Each sliver consists of:



- A use-case type requirement which contains the definition of a *user scenario* (a description of expected user activities), as shown on the previous page.
- A set of child test requirements stating how the system shall be tested for compliance with the use-case requirement.
- The actual tests (not shown)

The test requirement titles are simple statements of what shall be tested. The body of the test requirement (not shown) contains more detailed test design. Reading the titles is a very quick and powerful way to understand the test coverage, while putting the details of the test design in the body of the test requirement means we have all the test design information in one place.

The tests are then written and linked to the test requirements, so we can use the test requirements to index the tests. It is also possible to classify the slivers and the test requirements several different ways, so one can examine the coverage from different perspectives. The slivers and their child test requirements are organized by user activity to maintain a focus on users, but the test requirements are also are indexed by quality attributes (performance, reliability, etc) so we can assess coverage in this dimension.

Our tests had previously been stored in home-grown test management systems, which stored only the tests. When we converted our tests from their earlier test-only format into this requirement-and-test format, we reduced the number tests by an average of 25% *without reducing coverage at all,* simply because we were able to spot duplication of coverage and outmoded tests so easily. Other HP organizations have had similar results.

Now we had chunks which had user-value, and we had a way of detecting when a chunk was done. We were ready to start Lean Product Development – running the chunks through a development system.

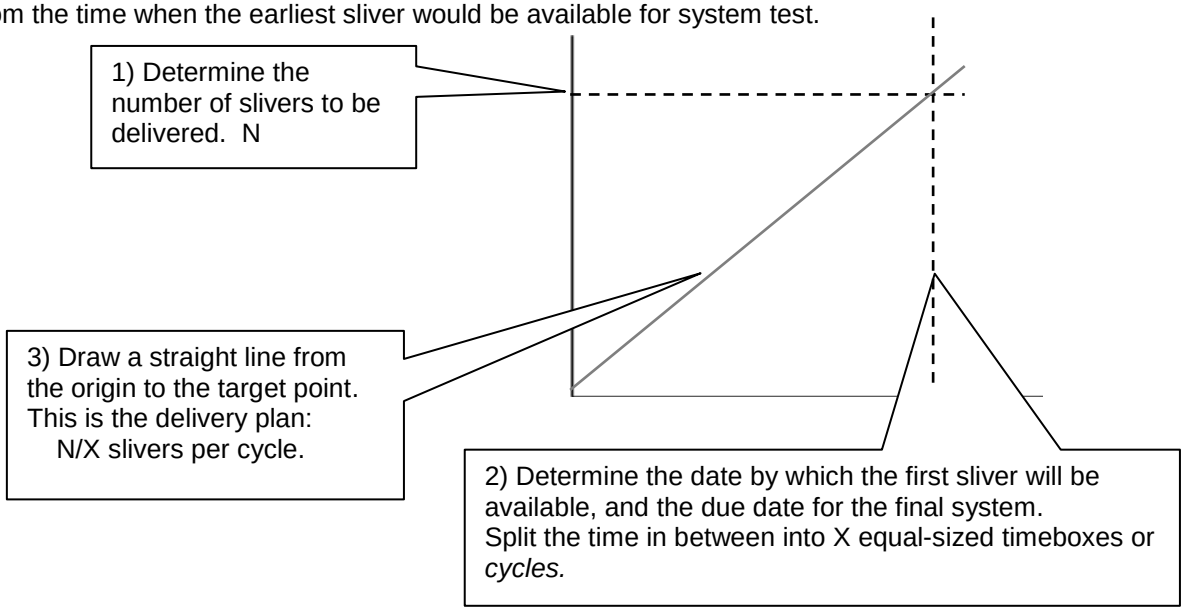
Setting up a Schedule – Session Planning

Now that we had the work broken into slivers, Lean and agile principles both suggest that the least expensive way to deliver the slivers would be to deliver them gradually throughout the program, rather than all at once near the end.

Most of the subsystem teams were already doing some form of agile, so the idea of delivering in chunks wasn't foreign to them. None of them had the same set of chunks, however. We presented our slivers as a reasonable *system-level* set of chunks, which would be common to all organizations during system test even though they still had their own individual chunks during component and subsystem development. This meant each organization had to create some sort of mapping between their chunks of development and our slivers. We had around a hundred slivers at this point, many of which were almost entirely legacy features.

The Marketing and Customer Satisfaction team members usually found that the slivers directly represented the features in which they were interested, and were eager to see if this method would lead to reporting progress of individual features.

We presented the simplest possible plan to the development teams: a linear delivery of slivers, starting from the time when the earliest sliver would be available for system test.



The different organizations had their own different-sized time boxes starting on different calendar dates, so our organization defined our own cadence of four-week cycles called *sessions*. The length was based on the amount of time we thought it'd take to run all the system tests, allow one round of defect fixes to be coded and integrated into the item under test, and verify the fixes. A four-week cycle gave a sliver a fighting chance to pass its tests and reach "Completed" in a single cycle.

The decision of which slivers would be delivered in any particular time box was negotiated with the technical leads of all subsystems. Most of the slivers included a list of "features needed from development before running this sliver", so it was possible to have reasonable (although long) conversations about what order would best accommodate all the different subsystem teams' needs.

We prepared our "session plans" showing which slivers would be tested in which timeboxes, and the technical leads compared the planned output of their time boxes with our sessions and slivers. Where our plan called for testing something that wasn't developed yet, either we moved the sliver later or the development team swapped their order around until we had a reasonable fit. Most teams didn't do a formal mapping between their chunks of development and our slivers, so this comparison was done on the fly.

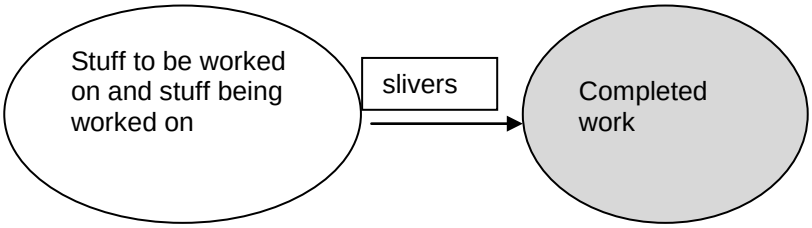
Once we had a final "session plan" which was agreed to by all partners, we were off to the races.

Monitoring Progress – the Cumulative Flow Diagram

Because of the enormous number of people and organizations involved, it's very easy for programs like these to get off track. A key aspect of Lean Product Development, and one that proved invaluable to us, is the emphasis on monitoring the performance of the *entire system* first, and examining individual pieces of the system only as needed. In our case, the entire system consists of the development teams ***and*** the integration & system test team. The intent is to put saleable features on the shelves for HP, and that takes all of us working together.

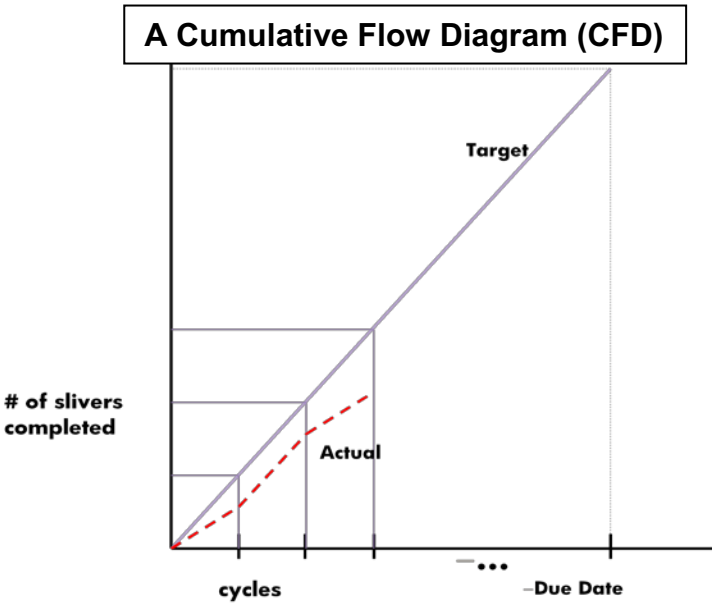
Lean Product Development tells you to draw a model of your system, showing activities and the flow between them. The model in turn tells you what to measure in order to monitor the overall progress. Typically one starts with a high-level, simple model.

At the highest level, the system looks like this:



This simple model is monitored by a *cumulative flow diagram (CFD),* which shows the actual number of slivers completed in each cycle.

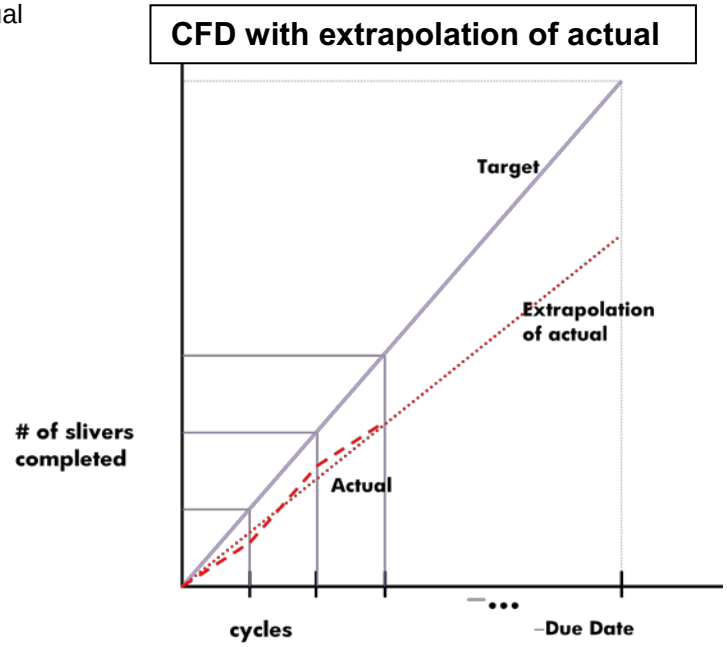
Notice that we aren't showing how many *tests* are passing – we are showing how many *features* are passing testing. This is a more direct measure of the final, saleable output of the entire system.



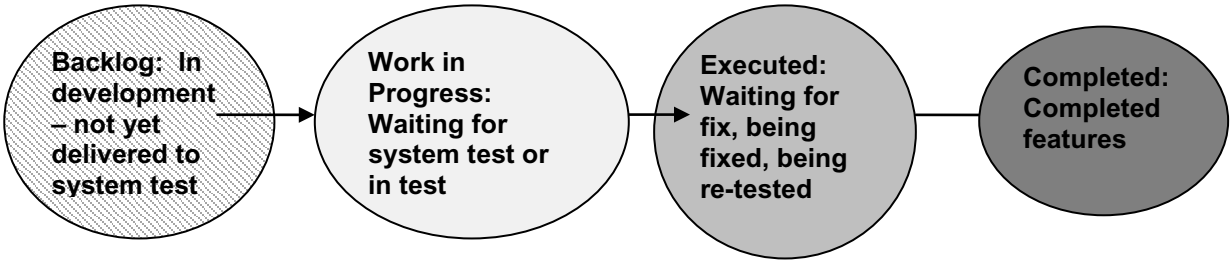
In this CFD, a simple extrapolation of the actual progress is possible after three or more cycles. This extrapolation clearly shows discrepancies between the plan and the predicted actual.

Any time fewer than the target number of slivers is delivered in a cycle, the program has fallen behind schedule. The following cycle will need to "catch up" by delivering more than the target in the next cycle.

So far, all the large programs where we used this method were behind schedule within the first few cycles. Some program teams found this difficult to accept at first and argued vociferously that the measurements must be incorrect, while others said "I thought we were behind, but now I know how much".

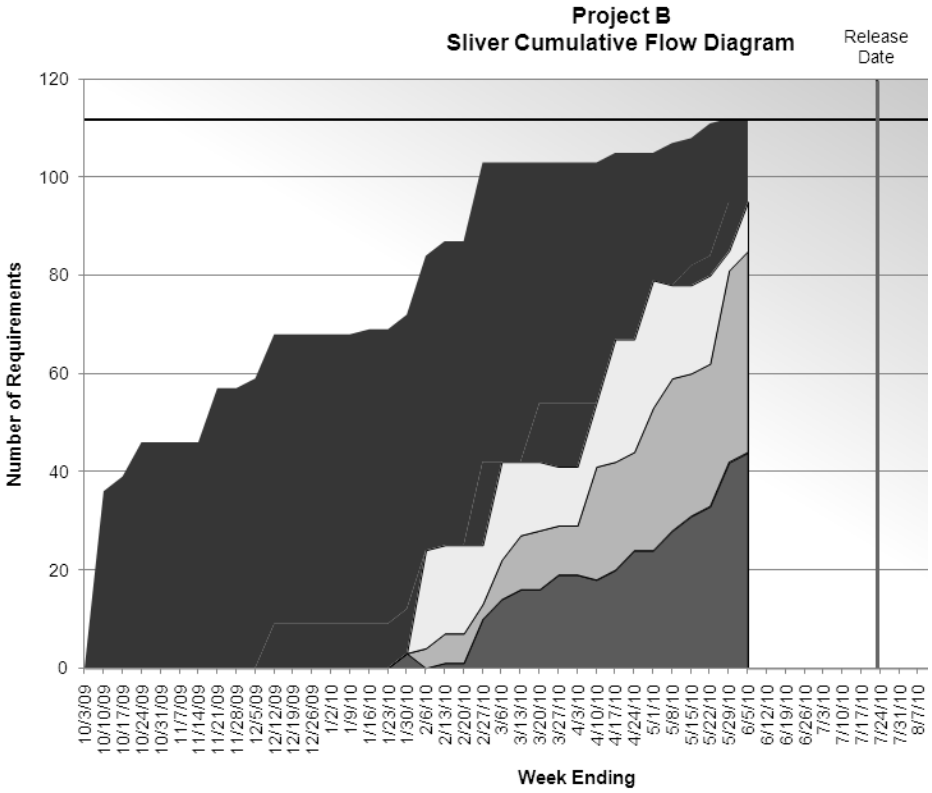


The first step in changing the throughput of any system is to understand why the throughput is what it is. This usually requires a more sophisticated model than the simple model on this page. The most frequently asked questions were around whether the problem was in System Test or in one of the Development organizations, so we broke up the first bubble in the previous model into three bubbles, showing the interplay between test and development.



Each sliver has a status flag, which can be set to any one of the statuses shown in the model below. The dates on which each sliver moves from state to state are tracked in QualityCenter. We wrote a program to extract the data from QualityCenter and graph it as a cumulative flow diagram.

This is a typical CFD.



Key:

	Under Development
	In Test
	Being Fixed
	Completed (All Tests Pass)

With the slivers, their tests, and the CFD metrics program in place, we were ready to run a full program with Lean Product Development.

Project B runs under Lean Product Development

Project B is a high-end laser program similar to Project A. We started discussing the sliver-based system testing with this program about half-way through its development, well before system testing would usually start. The various organizations and the overarching program team were both willing to change their planned deliveries slightly to accommodate the sliver model.

As usual, the teams were eager to start system testing early. However, during the first 4-week session of system test, almost no slivers passed all their system tests.

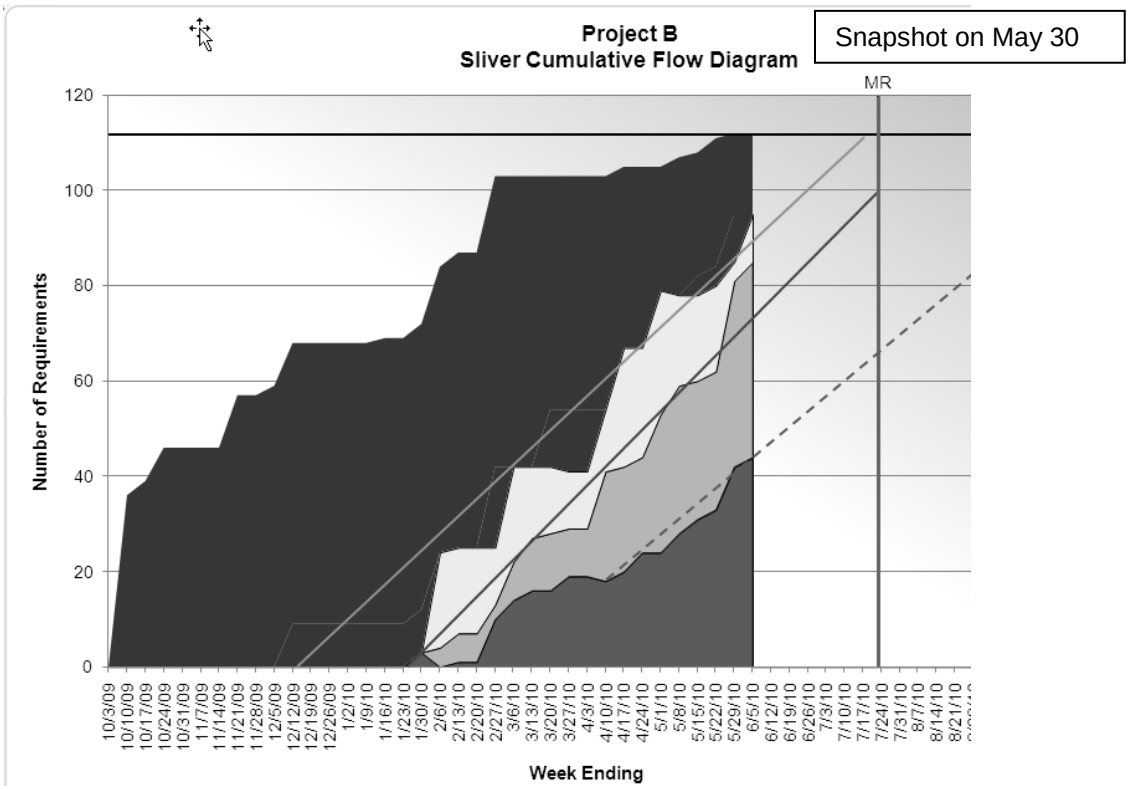
A quick look at the defects found per test in the first session explained what was going on. The first session starts with an entrance or acceptance test suite, assessing the capability of the system to be system-tested. Nearly every test run in this suite discovered a defect, many of which were attributed to necessary code not having been delivered to system test.

This may sound silly – unknowingly testing features which haven't been completed – but remember that these teams had been waiting until all features were complete before running these high-level system tests for any feature. Up to this time, there hadn't been a need to track exactly when all the work pertaining to a particular feature was completed across dozens of individual teams, so there wasn't a good high-level tracking system in place. Informal tracking didn't work well because the work could be spread across three separate organizations comprising hundreds of people.

Once the short entrance suite had been run, the test manager looked at these results and stopped the system testing. This forced the development organizations to discuss among themselves how they would know when a feature was ready to test, and they quickly improved their ability to track this. Stopping the testing also saved a good bit of money which would otherwise have been wasted on running tests which would repeatedly report that the same features weren't finished.

In the next cycle, a number of slivers were ready to test and the testing proceeded, finding and reporting useful defects to the various development teams. The bulk of the issues quickly changed from un-delivered functionality to mostly defects. The test leads showed the CFD every week to the leads of the various development teams.

Things proceeded along for some weeks. Each week, the test leads reported which slivers weren't completed as planned, and what needed to be delivered from R&D in order to close these particular slivers. The CFD slowly showed a divergence between the plan and the extrapolation of the actual, indicating that the program was falling behind schedule.



On May 30, the sliver CFD showed:

- The rate of slivers delivered *into* system test (gray line) and the rate of slivers *finishing* the first round of system test (dark line) are essentially the same.
  → This means that testing is keeping up with development.
- The rate of slivers *finishing the first round of system test* (dark line) is far higher than the rate of slivers *entering the Completed state* (dotted line).
  → This means the fixing and re-testing isn't keeping up with either development or testing.
- The dotted line showing extrapolation of the Completion rate crosses the Target Release vertical line well below the target number of requirements
  → This means that the last sliver isn't going to complete until *well after* the target MR date.

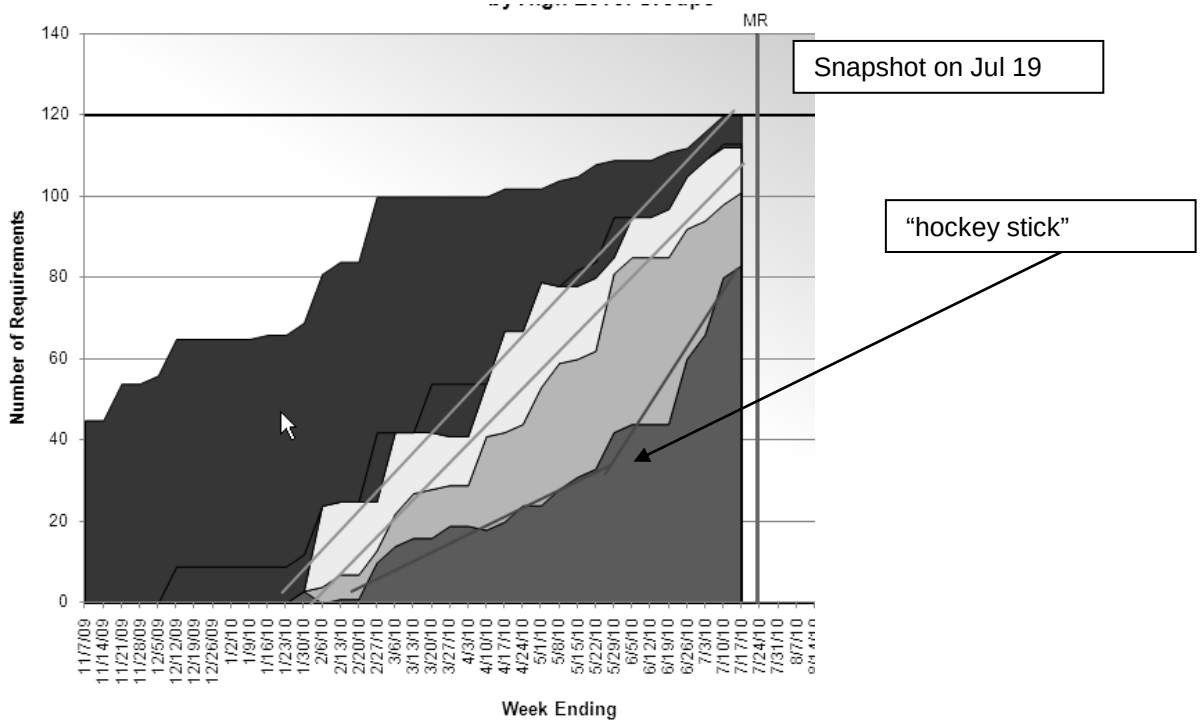
Examining the open defects, it was obvious that the main problem was a large number of unfixed defects. The program manager was now monitoring the status of hundreds of defects, some of which were more than two months old. The program manager needed to track when they would be fixed, but also:

- What else is this defect blocking?
- What parts of the system are affected?

Even a couple dozen such defects is quite a lot to carry in your head. We found that the association of defects with slivers made it much easier for the program manager to understand and remember what was blocked and what was affected. Monitoring the status of slivers, rather than tests per se, also allowed the teams to target groups of defects which were blocking or affecting the same area of the system, and get the area sufficiently fixed to allow system testing to take place. This reduced the overall complexity of managing the project.

Of course, the reduction in complexity, while pleasant, didn't fix the divergence between planned and actual. The program was still behind schedule. The program team started to say "we need to see a hockey stick" in the CFD – that is, a sudden and sustained turn upwards. As usual, saying it didn't make it happen. After a couple of weeks of watching the growing divergence between the planned and actual,

the program teams decided to take action. They stopped working on new features for a full week and spent all their effort on fixing defects. The defect fix rate spiked sharply, but what we were really watching was the hockey stick in the CFD – and sure enough, it appeared.



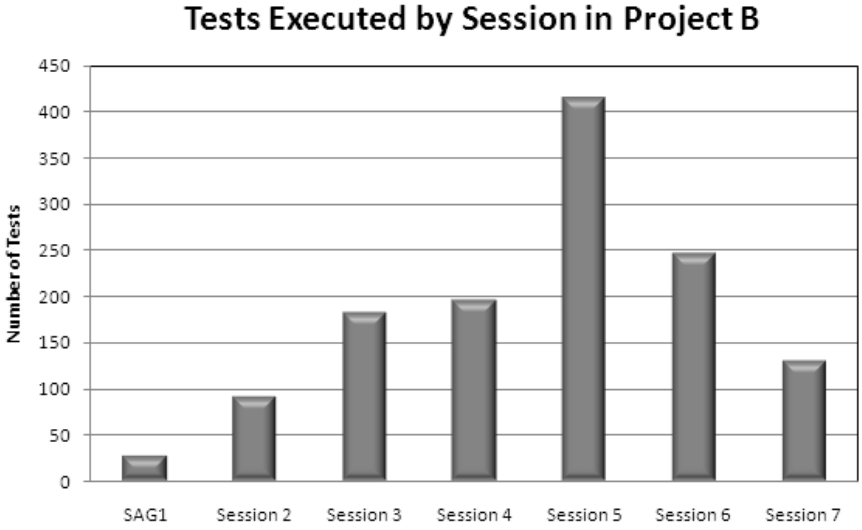
The large collection of defect fixes allowed a substantial number of slivers to re-enter testing and complete almost immediately. With the backlog mostly gone, defects found in subsequent slivers were fixed more quickly and the more rapid progress rate was sustained through the next months.

The End of Project B

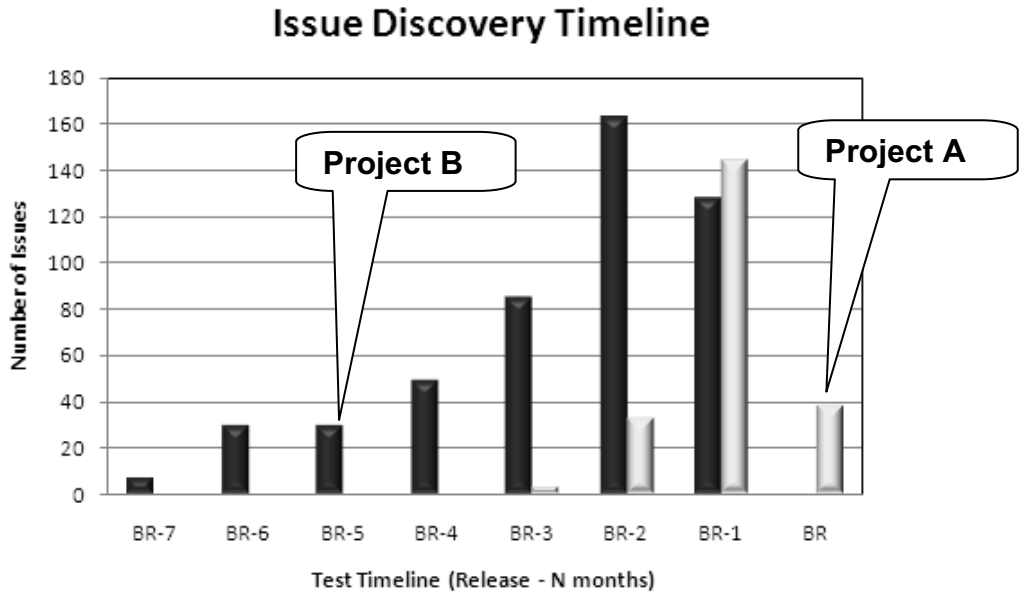
As Project B nears its end, it is apparent that we didn't really achieve a smooth, linear delivery of features over the length of the project.

This chart shows tests executed by session in Project B.

Clearly the testing wasn't evenly spread throughout the seven months of system test.



However, we did see a very substantial change in feedback time between Project A and Project B. As seen in the defect discovery timeline, Project B started system test seven months before final release, compared to Project A which started three months before final release. This is definitely a substantial improvement in feedback time, since some developers received feedback three to five *months* (not weeks) earlier than they did in Project A.



Conclusion

We've really just begun our transition to Lean Product Development. The initial steps of the transition were quite costly, since we had to reorganize our system tests around features (as opposed to capabilities), *and* organize them in a way that would correspond to recognizable user stories, *and* organize them in a way that would not require enormous changes in the order in which work was done in each of the numerous subsystem teams. However, we got some immediate benefit due to the 25% reduction in the number of tests, so we had some time and resources to use on the transition.

Now that we have the foundation built and have started using Lean methods, we find that we can:

- Start system testing three to four months earlier than previously, thus cutting feedback time dramatically for half the defects.
- Show convincingly and credibly that a program is behind schedule and the improbability of catching up while still doing everything the same.
- Allow a program manager to understand quickly what features are blocked by which defects and thus simplify managing the program.
- Convince a program team to stop coding and fix for a week, despite the huge inertia inherent in large groups of teams.

However, we haven't yet started applying Lean to a program early enough to result in an efficient and well-understood order of development and integration across all the many subteams. We need to work out the order in which slivers will be delivered much sooner, near the very beginning of the program. This will provide a framework for the subsystem teams to align with each other and plan to deliver some features in full much earlier.

We're optimistic about achieving radical improvements in productivity next year as we continue to implement the Lean Product Development methods.

References

[Anderson 2004] Anderson, David J. 2004. *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results.* Pearson Education

[Anderson 2010] Anderson, David J. 2010. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press; Sequim, WA, United States

[Jeffries 2001] Jeffries, Ron; Anderson, Ann; Hendrickson, Chet. 2001. *Extreme Programming Installed*. Addison-Wesley.

[Poppendieck 2007] Poppendieck, Mary; Poppendieck, Tom. 2007. *Implementing Lean Software Development: from Concept to Cash*. Addison-Wesley.

[Reinertsen 2009] Reinertsen, Donald G. 2009. *The Principles of Product Development Flow: Second Generation Lean Product Development,* Celeritas Publishing. Redondo Beach, CA, United States.

Lean Product Development Bibliography

Anderson, David J. 2004. *Agile Management for Software Engineering: Applying the Theory of Constraints for Business Results.* Pearson Education

Anderson, David J. 2010. *Kanban: Successful Evolutionary Change for Your Technology Business*. Blue Hole Press; Sequim, WA, United States

Reinertsen, Donald G. 2009. *The Principles of Product Development Flow: Second Generation Lean Product Development,* Celeritas Publishing. Redondo Beach, CA, United States.

Kniberg, Henrik and Skarin, Mattias. 2010. *Kanban and Scrum: Making the Most of Both.* C4Media.

Software Product Lines Bibliography

- Northrop, Linda and Clements, Paul. 2001. *Software Product Lines: Practices and Patterns.* Addison-Wesley.
- Software Engineering Institute at Carnegie Mellon: Software Product Lines http://www.sei.cmu.edu/productlines/
- www.softwareproductlines.com

Note: Software Product Lines is sometimes called Software Product Families in Europe.

In Case You Wondered....

Lean Product Development isn't identical to Lean Manufacturing. They both use the same fundamental concepts from queuing theory, but manufacturing deals mainly with predictable tasks of similar sizes and relatively low variability, whereas product development deals with tasks which are inherently variable and dissimilar. Since the nature of the tasks is different, the application of queuing theory sometimes leads to different answers for Lean Manufacturing and Lean Product Development. For more on this subject, see Reinertsen 2009.

Bridging the Cultural Gap

Katherine Alexander
Test Engineer, Vertex Business Services
Katherine.alexander@vertexna.com

Abstract

The ever-increasing globalization of the workplace, and subsequent geographical dispersion of employees, creates many challenges for today's testers, including communicating across different time zones, overcoming language barriers and maintaining a consistent team framework. One unique challenge that should not be overlooked is that of cultural diversity and the stereotyping that can and does occur. Whether it's Americans working with East Indians, Europeans working with Asians, or even east and west coasters in the United States, there are cultural differences that might result in participants hindering a project inadvertently. Because perceptions affect interactions, the negative results range from simply frustrated communication among team members to delayed projects.

These obstacles need not be a hindrance to the testing process. Learn what the common stereotypes are and how to overcome pre-conceived notions, devise efficient work practices and enjoy the benefits of creating a more cohesive testing team across the globe.

Biography

Katherine Alexander graduated with a Bachelor of Arts Degree in Psychology from California State University, Fullerton with an emphasis on interpersonal relationships. She has been working as a Test Engineer since 2006 and is currently employed with Vertex Business Services where she focuses on custom projects, collaborating with the client to discover solutions that meet their business needs. The recent global expansion at Vertex Business Services and the changing demands of her current position has prompted a renewed interest in her previous studies and research.

Introduction

Globalization in the workplace and geographically-dispersed teams are here to stay. There are benefits to the global office – employees can work from anywhere at any time which aids in continuing iterative development and testing on a project. People can be hired based on their competencies, not just their physical location, enabling managers to build teams with higher skills.

There are also challenges to having team members spread across the globe. Differing time zones can make communication difficult. Email is an effective form of communication but sometimes a phone call can resolve a problem more quickly. Language barriers can also present difficulties. Probably the greatest challenge is that of cultural diversity. Each culture has its own social norms and customs that dictate how the individuals in that culture approach their work and their coworkers. It is important not

to overlook this aspect or fall into the trap of stereotyping certain behaviors; instead, it is important to understand these behaviors and be better able to communicate with all members on the testing team. Delivering the best quality products in the most efficient timeframe is the central goal of all team members, and basic cultural literacy can better equip the team to accomplish its goal effectively.

What is Culture?

We first need to understand culture and why it varies so much among groups of people.

There are three layers of culture (O'Neil, 2006). The first layer is the body of cultural traditions that distinguish a specific society. This includes shared language, traditions and beliefs that set these people apart from others. This is what we generally refer to when speaking about Italian, Japanese or Indian cultures.

The second layer is what is known as a subculture. In complex, diverse societies in which people have come from many different parts of the world they often retain much of their original cultural traditions, sharing a common identity, food tradition, dialect or language that sets them apart from the rest of their society. Examples of easily-identifiable subcultures in the United States include ethnic groups such as Vietnamese Americans, African Americans and Mexican Americans.

The third layer consists of cultural universals: learned behavior patterns that are shared by all of humanity. No matter where people live in the world, they share these universal traits. Examples of these traits include distinguishing between good and bad behavior, using verbal language with a limited set of sounds and grammatical rules, raising children in some sort of family setting and establishing leadership roles for making community decisions.

All cultures exhibit these traits but have developed their own specific ways of carrying them out. For example, people in deaf subcultures use their hands to communicate with sign language instead of verbal language. However, sign languages use grammatical rules the same as verbal languages.

Let's look at an example of why it is so critical to understand other cultures that are working together. When negotiating in Western countries, the objective is to work toward a target of mutual understanding and to "shake hands" when an agreement is reached. This is the cultural sign that negotiations have ended and working together can begin. In Middle Eastern countries, shaking hands is a pre-cursor to negotiations: this is the not the sign that the deal is complete, but that serious negotiations are just beginning (Itim, 2009).

In testing, it is important that all team members understand the requirements and test plan of a project and that this is communicated clearly between the tester and the developer at the outset of a project rather than halfway through. This will avoid any delivery delays or out-of-scope issues.

Cultural Dimensions

Dr. Geert Hofstede conducted what is probably the most comprehensive study of how values in the workplace are influenced by culture (Roberts, 2008) and his results have become the international standard by which to implement global Human Resources policies (Higgs, 2005).

Beginning in the 1970's, while working at IBM as a psychologist, Hofstede collected and analyzed data from over 100,000 individuals in over 40 countries. His research was done on IBM employees only, which allowed him to attribute the patterns to national differences in culture rather than differences in company culture. From these findings, he initially identified four distinct cultural dimensions that served to distinguish one culture from another (Mind Tools, 2010). He later added a fifth dimension which is how the model is presented today.

The five dimensions are:

1. Power Distance

Power Distance Index (PDI) focuses on the degree of equality, or inequality, which exists and is accepted between people in a country's society. A higher score indicates that society accepts an unequal distribution of power and people understand their place in the system. These societies are more likely to follow a caste system that does not allow significant mobility of its citizens. A low power index indicates that power is shared and well-dispersed, that society members view themselves as equals.

Characteristics of a high PDI are centralized companies, with strong hierarchies and large gaps in compensation, authority and respect. The leader's power is acknowledged and answers come from the top. A low PDI is characterized by flatter organizations where supervisors and employees are considered almost as equals. Teamwork is relied upon and many people are involved in decision making.

In a country with a high PDI, such as Malaysia, one would probably send reports only to top management and have closed door meetings where only a select few, powerful leaders were in attendance.

2. Individualism

Individualism (IDV) relates to the degree the society reinforces individual achievement and interpersonal relationships. A high IDV ranking indicates that individuality and individual rights are paramount within the society. There is a loose connection between people. A low IDV ranking demonstrates societies with a more collective nature and close ties between individuals. These cultures reinforce extended families and collectives where everyone takes responsibility for the group.

High IDV is characterized by high valuation on people's time and their need for freedom, an enjoyment of challenges and an expectation of rewards for hard work, a respect for privacy and an encouragement of debate and expression of individual ideas. Characteristics of low IDV scores are: an emphasis on building skills and becoming master of something, harmony is more important than honesty, respect is shown for age and/or wisdom and change is introduced slowly.

In applying Hofstede's analysis, we understand that, in Central American countries like Panama where the IDV scores are very low, a marketing campaign that emphasizes benefits to the community or that is tied into a popular political movement would likely be understood and well-received.

3. Masculinity

Masculinity (MAS) refers to the degree the society reinforces the traditional masculine work role model of male achievement, control and power. A high MAS indicates the country experiences a high degree of gender differentiation. In these cultures, males dominate a significant portion of the society and power structure, with females being controlled by male domination. A low MAS ranking indicates the country has a low level of differentiation and discrimination between genders. In these cultures, females are treated equally to males in all aspects of the society.

High MAS scores are characterized by men being masculine and women being feminine, a well-defined distinction between men's work and women's work. Men are advised to avoid discussing emotions or making emotionally-based decisions or arguments. In low MAS societies, powerful women are admired and respected, there is an effort to ensure job design and practices are not discriminatory to either gender and men and women are treated equally.

According to Hofstede's rating, Japan is highly masculine while Sweden has the lowest measured value. A testing team in Japan would have greater success with a male employee appointed to lead the team and a strong male contingent on the team. A Swedish testing team would aim for a balance of skills rather than gender.

4. Uncertainty Avoidance Index

Uncertainty Avoidance Index (UAI) focuses on the level of tolerance for uncertainty and ambiguity within the society, the level of anxiety members feel when in uncertain or unknown situations. A high UAI scoring nation has a low tolerance for uncertainty and ambiguity. This creates a rule-oriented society that institutes laws, rules, regulations and controls in order to reduce the amount of uncertainty. In low UAI scoring countries, there is less concern about ambiguity and uncertainty and more tolerance for a variety of opinions. This is reflected in a society that is less rule-oriented, more readily accepts change and takes more and greater risks.

Low AUI scoring cultures are characterized by very formal business conduct with many rules and policies, a need for and expectation of structure and avoidance of differences. A sense of nervousness spurns high levels of emotion and expression. Characteristics of low AUI scoring countries are: an informal business attitude, a greater concern with long-term strategy than what happens on a daily basis, an acceptance of change and risk, and an expression of curiosity when differences are discovered.

Argentina has a high UAI, so it would be better to investigate the various options of a project, present a limited number of choices, but maintain very detailed information on contingency and risk plans.

5. Long Term Orientation

Long Term Orientation (LTO) describes the degree to which a society embraces long-term devotion to traditional, forward thinking values. High LTO scores indicate the country prescribes to the values of long-term commitments and respect for tradition. Delivering on social obligations and "saving face" is considered very important. A low LTO ranking indicates the country does not reinforce the concept of long-term, traditional orientation. Instead, change occurs more rapidly as long-term traditions and commitments do not become impediments to change.

Characteristics of a high LTO are a strong work ethic, parents and men have more authority than young people and women, there is a high value placed on education and training and perseverance, loyalty and commitment are rewarded. In contrast, the characteristics of a low LTO are promotion of equality, high creativity and individualism, self-actualization is sought and there is no hesitation to introduce necessary changes.

People in the United States and United Kingdom have low LTO scores. This suggests that almost anything can be expected from these cultures in terms of creative expression and new ideas. Tradition is not as valued as in other countries so people in these societies are likely willing to help execute the most innovative plans as long as they can participate fully.

Benefits of Hofstede's Work

Often businesses and even testing teams see cultural diversity as an area of difficulty and obstacle rather than an opportunity to build a competitive advantage. Laurent and Adler (1983) carried out an exercise that illustrates this point well. International executives attending management seminars in France were asked to list the advantages of cultural diversity for their organizations. 100% of participants were able to identify disadvantages while less than 30% could identify any advantage.

A common, stereotypical fear regarding globalization is that another country will "steal" our jobs from under us. However, when examining data on changes in the U.S. work force, Greenwald and Kahn (2008) show that this isn't the case, that job losses due to higher productivity, often the result of improving technology, greatly outnumber those lost to globalization. In fact, the U.S. Department of Commerce estimates that 65% of job losses in manufacturing between 2000 and 2006 were due to productivity increases while just 35% of job losses owed to overseas outsourcing. It can be assumed that those in the software industry hold the same types of fears and misconceptions.

This preconceived notion is most probably caused by the lack of understanding we generally have about other cultures. That which is not easily understood or familiar can become a roadblock. Hofstede's work removes these roadblocks and provides a starting point for conversation between different groups. He has provided scores for 57 countries that depict the dimension scores for that country and culture with an explanation of how they uniquely apply to that country (geert-hofstede.com, 2009). These scores provide a guide for how to approach a different culture.

For instance, when working with a low PD scoring culture, acknowledge a leader's power. This will be the person to go to when decisions are needed on testing procedures. When differences are being discussed with a high LTO culture, show respect for traditions and avoid doing anything that would cause another to "lose face".

Criticisms of Hofstede's Cultural Dimensions

As useful as Hofstede's Model of Cultural Dimensions can be, there are a few drawbacks that one should keep in mind (ClearlyCultural.com, 2009).

First, the whole averages of a country do not equal the individuals of that country. Hofstede's model has proven to be quite accurate when applied to the general population, but it should be remembered that not all individuals or even regions within subcultures fit the description. There are always exceptions to the rule, so use the model as a guide to understanding differences rather than a law set in stone.

Second, there is a question of how accurate Hofstede's data is. It was collected through questionnaires, which have their own set of limitations. In addition, the context in which a question is asked in some cultures is different from its content. Especially in group-oriented cultures, individuals tend to answer questions as if they were addressed to the group the individual belongs to rather than the individual herself.

Last, how up-to-date is the data? Hofstede's research was conducted in the 1970's. How much have some of the cultures studied changed since then? How much impact has technology had in recent years? Technology has brought the world much closer together over the last decade and different cultures have found themselves working together and communicating more since Hofsetede's

differences were established. Furthermore, cultures are becoming mixed through foreign studies, immigration and foreign postings. It is quite common these days to meet an Indian who behaves very American because he has spent so many years studying in the United States.

Additional Steps to Bridge the Gap

Recognizing that cultural differences exist and respecting traditions is a good start to bridging the cultural gap in the workplace. There are several other simple activities that testing teams and team leaders can implement to make communication easier and more effective, and to build a more cohesive team overall.

As previously mentioned, language barriers can and do impede communication. In the United States, the majority of workers speak English, which is one of the hardest languages to learn. There are many words that are spelled the same but have different pronunciations and different meanings. Americans tend to use a lot of slang, which is confusing to non-English speaking people. Avoid the use of slang when communicating with different cultures. Use e-mail when appropriate, as written communication removes accents that might be difficult to understand and gives the author and reader the ability to re-read the text, to gain a better understanding of what is trying to be communicated.

In addition, speak slowly and clearly. Even when rushed for time, don't rush through communication. It is better to take the time than risk miscommunication that results in additional time to clear up the confusion. Ask for clarification and don't assume understanding of what has been said. Frequently check for understanding and rephrase what has been said when necessary (Culturosity.com, 2007).

When possible, it is very helpful and beneficial to send team leaders to visit the corresponding locations. This allows a brief immersion in the culture that can provide a greater understanding of customs and behaviors. Each culture is able to teach the other about itself and the in-person meeting provides the type of bonding that isn't available to groups working at great geographical distances, thus enforcing the idea of team and the formation of a group with the same purpose (Outsourcing Factory Incorporated, 2008).

One of the most popular processes used in testing today is the Agile process which is very conducive to working within geographically-dispersed teams. Especially between different time zones, iterative development allows testers to work on each piece as it's developed and allows quick turnover in solving issues found. It also provides frequent checks that testing is proceeding as according to plan and minimizes risks. The frequent communication needed provides more contact and a better understanding of the process between differing cultures, thus building a base for future projects (CyberMedia, 2010).

Closing the Gap

Being a software tester in today's globalized marketplace requires an increasingly-high level of interpersonal skill and cultural savvy. With the proper training of team members and the recognition of the differences that exist between cultures, globalization can enhance the testing process. Each individual brings unique experiences, knowledge and ideas to the group which can have a positive impact and contribute to the success of testing projects. Great success can be achieved by being aware of the cultural gap. Even greater success can be achieved by putting in the time and inter-personal research necessary to bridge it.

Performance Testing and Improvements Using Six Sigma – Five steps For Faster Pages On Web and Mobile

Mukesh Jain
Mukesh.Jain@Microsoft.com

Abstract

Through this paper, I intend to highlight the most important part of any business – the end-user. Without the end-user, the financial equation can never be balanced. The best way for any organization to remain profitable and competitive is to keep the end-user satisfied and happy. Understanding end-user expectations with our software + service and driving that back into our development processes to create appropriate solutions that satisfies the end-users is key for any product or service to be successful.

In this paper, I will describe a Six Sigma based improvement model that enabled us to analyze the end-user experience, assess effectiveness of our services and make data-driven decisions in improving and sustaining service quality. I will also provide details on the QoS (Quality of Service) programs with which we at Microsoft have made significant quality improvements in online services like Bing, Hotmail & MSN.

Biography

Mukesh Jain *is currently Principal Test Manager in Microsoft Advertising R&D, leading multiple teams in USA and India to build next generation Search Advertising platform for Web, Mobile and Microsoft-Yahoo Search Alliance. Prior to this, he has been driving quality of service (QoS) strategy and breakthrough improvement in Bing, MSN, Hotmail & Messenger for USA, India, Japan and China. In last 11 years at Microsoft, he has shipped multiple releases of major products like Office, Exchange and Windows Live.*

*He is recipient of Microsoft's most prestigious award "**Gold Star**," for three consecutive years, Microsoft's Asia Pacific Leadership award (runner-up), Role-Model, Great People – **Leadership award**, Innovation Award, Solution Excellence, and several quality of service focus awards.*

*He has bachelor's degree in computer engineering and science and has 15 years of experience. He holds several certifications: Microsoft Certified Application Specialist, Microsoft Certified Standards Professional, TSP Coach, PSP Engineer & Instructor, CSTE, CSQA, CQM, CQIA, CQA, CTFL, CPD, CPE, Six Sigma Black Belt, Microsoft Office Specialist, ISO 9000 Auditor, MOF, ITIL and iSixSigma magazine's "**Best Six Sigma Black Belt**" award. He is author of book "**Delivering Successful Projects using TSP and Six Sigma**". He is currently writing book on "**Web Performance Improvement**".*

1. Background

Microsoft has evolved from Software to a Software + Service organization. During this period, technology has evolved and the delivery mechanism has changed from Cards → Disk → CD → DVD → Internet.

The end-user needs the service anytime, anywhere and on any device. The user expectation around service quality has changed and the bar is much higher now compared to the early days of the internet. Our online services are used across the globe. The end-user experience with our online services needs to be comparable to Global and local competitors (Google, Yahoo, Rediff, QQ, etc.). Service quality impacts user experience and potentially impacts operational cost, mind-share, market share and revenue. Turning the tide on service quality can have real impacts on bottom line and the top line of our business.

2. End User Expectation

Before we talk about service quality – we need to make sure we understand the concept from an end-user perspective. When we hear the word quality – the first thing that comes to mind is the testing process and bugs. In the Software industry, Quality is often measured in percent defect (bug) free. Service quality (a.k.a. Quality of Service or QoS) comprises of Performance, Availability and Reliability measures of the online services.

With more and more online services available at little or no cost, switching to another service becomes easier and the only organization that can attract and retain end-users are those that can **consistently** provide a high level of service quality and innovate to meet the growing needs/expectations of the end-users and global market dynamics.

The end-user needs and expectations change as new technologies and products are available in the market. Things that used to delight the end-user may no longer be a delight factor – they may become a "must-need" part. The quality bar is always set by the end-user and it is our responsibility to understand that and cater our software + service offerings. The product features along with the service quality drives customer satisfaction that in turn drives business results. Here is a diagrammatic view of the same:

Figure 1: Business Results thru product quality

3. End User Needs analysis

End-user needs are not all of the same kind and do not all have the same importance. Needs are different for different segment of users in different countries/markets. End-user needs should be classified into categories and prioritized to be able to focus your effort on the important things for satisfying different end-users in different market segments based on the relative priority of each segment's requirements.

There are five types of end-user needs, or reactions to product/service characteristics and attributes:

1. **Delight factors / Attractive needs**: These really make your product or service stand out from the others. This is the best way to get our end-user to love our products/service and promote it.
2. **More is Better**: These are the things that make the end-user happy. (E.g. larger mailbox size, faster page, always on etc.)
3. **Must Have**: This is the core piece of user satisfaction. If certain features / attributes are missing in the product/service - the end-user will never be satisfied. (E.g. Secure, available majority of the time, send/receive emails, etc.)
4. **Dissatisfiers**: The factors that cause your end-user not to like your product (e.g. slow pages, poor reliability, poor compatibility, etc.).
5. **Don't Care**: The factors that users do not care about. Hence their absence or presence does not impact end-user satisfaction.

Here is a diagram (Kano Model[3]) that depicts the above types of end-user needs using a mobile phone example:

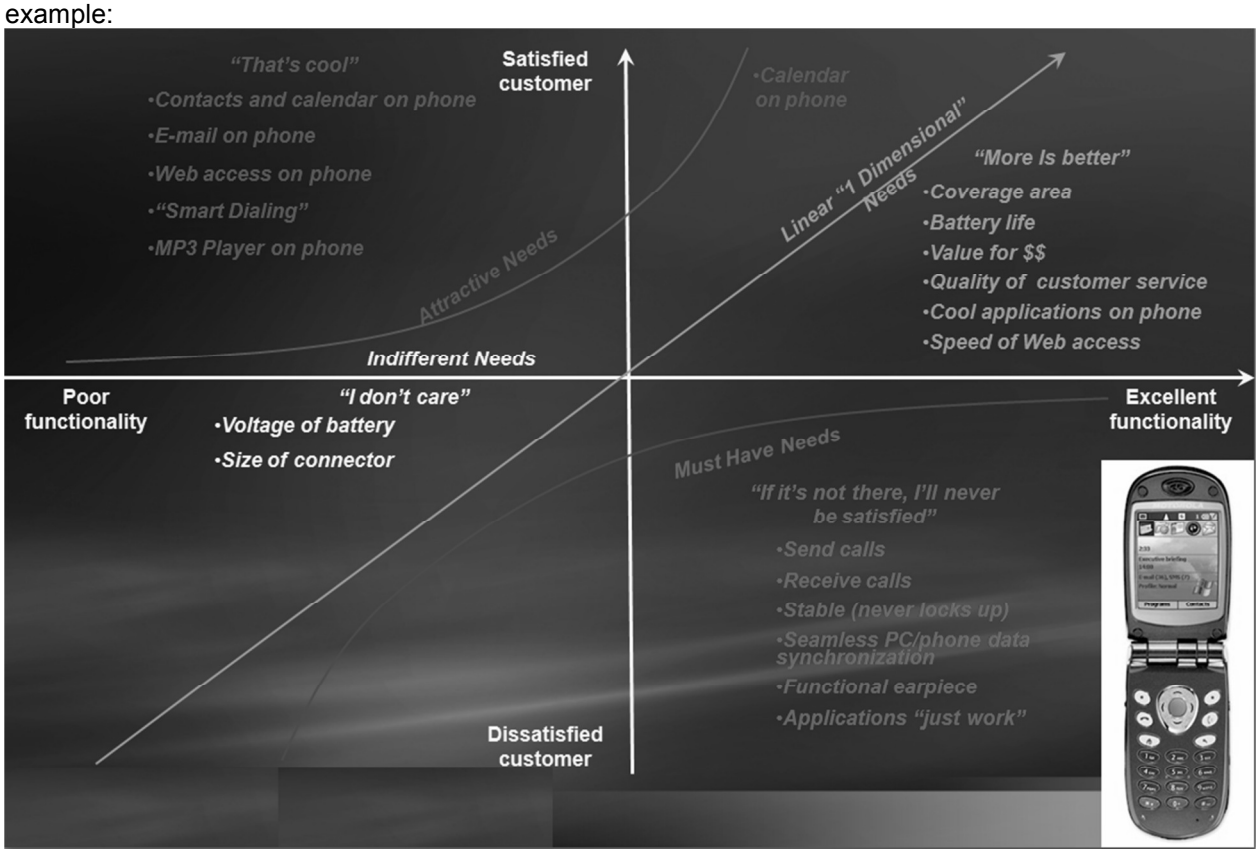


Figure 2: Kano Model showing end user needs

4. The Performance Issue

The data collected from customer satisfaction surveys and business metrics, it was clear that performance was the top concern for some of the services (Esp. Hotmail, Search and Mobile). We were losing user's mind-share in key markets (US, UK, Japan, India and China) resulting into lost revenue and market share. The obvious solution people came up with was to add more people to the team and do more performance testing and add more marketing budget to market the products. Based on my earlier experience using Six Sigma for breakthrough improvements, I suggested to senior Management to consider solving this issue in a more structured manner to that it can solve the problem and at the same time becomes part of the process. I got support from management to do a pilot project using Six Sigma and see if it helps. We started the project in late 2006 and involved key stakeholders (Engineering team, Operations, business folks, field marketing, management and support team).

5. Overview of Six Sigma

Six Sigma[1][2] is a combination of a philosophy, framework, methodology and roadmap for improvement and metrics. The goal of Six Sigma is to improve customer satisfaction by eliminating variation and defects from any process. Six sigma balances rigor and objectivity with speed and common sense.

The term Six Sigma is a statistical term for variation and it refers to the ability of a set of processes to produce the desired output (within specification limit). Specifically, processes that operate at six sigma level produce less than 3.4 Defects Per Million Opportunities (DPMO). E.g. In an telephone calling scenario, opportunity for defect could be calls dropped in the middle of conversation.

5.1 Solving the Right Problem – The Right Way

It is a fact that no software can be guaranteed 100% defect-free. When everyone feels that software will have defect no matter what we do, we stop taking action to prevent them from appearing. In fact, defects are inevitable. However, if we act that way, we make it horribly true. Simply fixing the bug is not enough. Take a moment to figure out why the bug happened. If we analyze our mistakes and re-think about the process, we can eliminate most of these mistakes. All processes have the potential for defects. Hence, all processes offer an opportunity for the elimination of defects and the resultant quality improvement.

Whenever you encounter an issue/defect, ask yourself:

- What assumptions were wrong?
- What rules did I break?
- How could I have detected this bug earlier?
- How could I have prevented this bug?

5.2 Finding and Fixing the Root Cause Using A Structured Approach

Detailed root cause analysis (using Six Sigma methodology) of all the high severity defects should be done on a regular basis (don't wait for project postmortem). Based on the findings of the analysis, appropriate process needs to be updated to ensure these kinds of defects do not occur and if they occur, can be catch early. This will help the team to prevent these defects in future. When everybody starts sharing this data with other people, all the teams will be in a situation of preventing majority of the defects – hence, doing it right the first time and shipping high-quality product on time.

There are multiple six sigma methodologies; out of them the most popular one is DMAIC[4] (Define, Measure, Analyze, Improve and Control). The key steps in the Six Sigma improvement framework are Define – Measure – Analyze – Improve – Control. When a specific Six Sigma project is started, the goals related to customer satisfaction should be established and further divided into sub-goals like cycle time

reduction, cost reduction, or defect reduction. (This can be done some of the tools in Six Sigma methodology).

Define (D) : Zero in on specific problem with defined return on effort
Measure (M) : Determine current performance of process
Analyze (A) : Validate key drivers of performance (root cause of problem)
Improve (I) : Improved performance and validated realized results
Control(C) : Implement controls to ensure continued performance

The Six Sigma project improvement team works together, brainstorms and identifies relevant metrics for the process and product, based on engineering principles and models. With detailed data on processes in hand, the team can then proceed to evaluate the data for trends, patterns, causal relationships, special cause, common cause, root cause, etc. Usually the majority of analysis for typical green belt projects can be done using a simple statistical tool. For more complex projects, you can do some experiments and modeling to evaluate/confirm hypothesized relationships or evaluate the degree of influence of one factor over another in a process.

It is often necessary for the teams to iterate through the Measure-Analyze-Improve steps until they find the right set of root cause and are able to fix a majority of them. When the expected goal is achieved, control measures are then established in the process to sustain improvements.

When using Six Sigma, it is important to identify which process measures significantly contribute to the overall customer satisfaction. I.e. what are the vital few that impact the customer satisfaction? The process and product quality is usually more sensitive to some factors than others. The analysis phase of Six Sigma can help identify the extent of improvement needed in each process sub-step in order to achieve the expected target/goal in the final product.

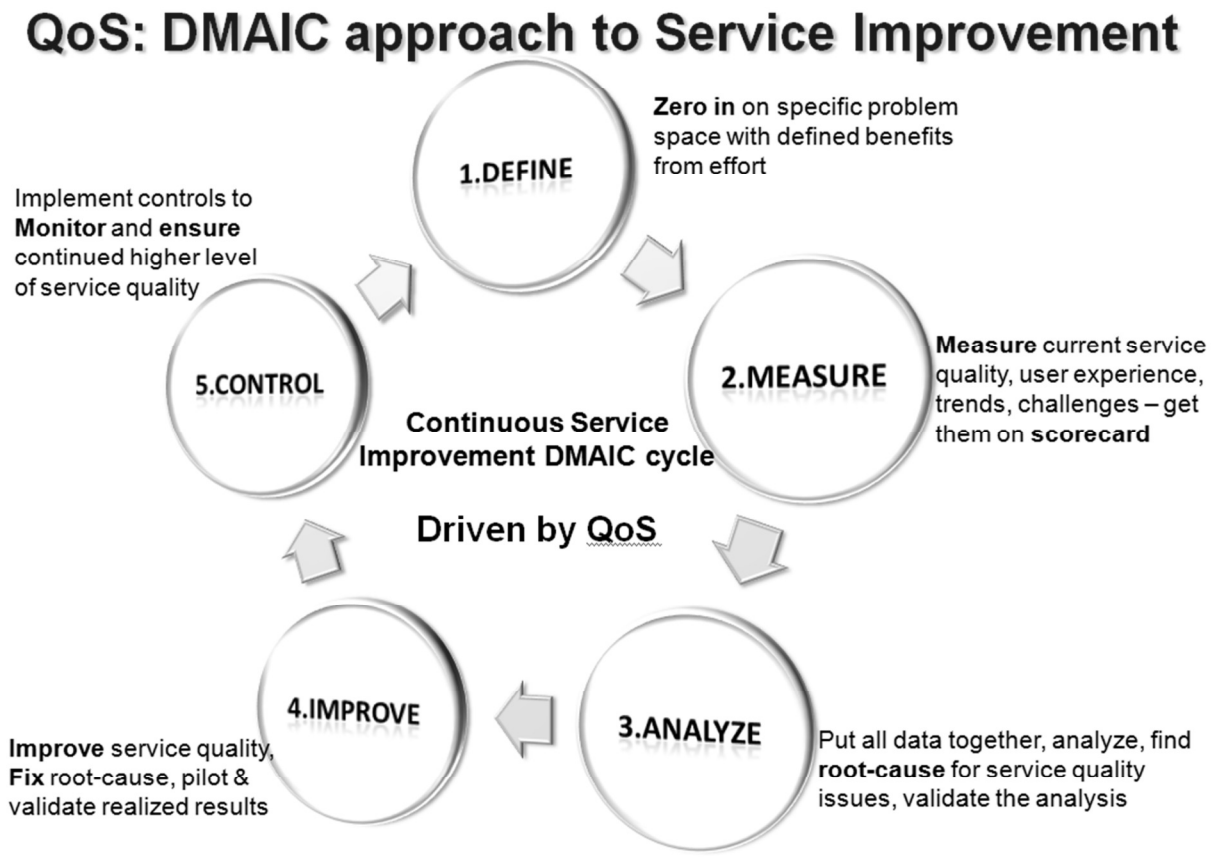


Figure 3: QoS: DMAIC approach to Service Improvement

5.3 Defining the problem and improvement project

The Define stage for the six sigma project calls for baselining and benchmarking the current process. It also includes an assessment of the organizational/group change that might be needed for success. Once an effort or project is defined, the team methodically proceeds through remaining phases of Six Sigma - Measurement, Analysis, Improvement, and Control.

The key deliverables of the Define phase is identify and form a committed cross-functional team along with support from key stakeholders to work on the improvement project. The team works on creating the project charter that defines the goal, high impact characteristics that are critical to quality (CTQs) and business process mapped. The charter details the business case for taking on this improvement project, defines the problem to be solved, timeline and roles of each team member. This is an important step and sets foundation for any improvement project.

5.4 Measuring Service Quality / End-User experience

End-user experience with our products and service can be measured in terms of Availability, Performance and Reliability of the service and reported on a dashboard on a regular basis. These data collectively represent service quality metrics. These data can be used to understand opportunities for improvements in our services and help us in driving product/services features and improvements.

What can service quality measurements tell us about the true end-user experience? How do you measure, what do you measure? There are multiple ways to measure and track these metrics. Some of the measurements are straight forward and others can be derived from them.

As software + service offerings become increasingly complex (Geo-Distributed, Global Caching, AJAX application, etc.) and end-user expectations for the service quality increases, predictable service quality becomes even more business critical. As we move toward an on-demand infrastructure, three important requirements becoming apparent:

1. Monitoring critical service capacity in real-time and managing it effectively is extremely important in order to maintain and improve service availability and response time to end-users. Users get frustrated if the service is not available, reliable or if it is slow. They may stop using the service and move to a competitor's offering.
2. Proactively identifying potential issues quickly and accurately by providing actionable data before they become serious problems is crucial to maintaining the service quality level and retaining the user base.
3. Metric methodology must be practical to implement to ensure consistent service monitoring & reporting solutions and integrate with existing infrastructure seamlessly without adversely affecting the service.

The focus on the measurement program is to be result-oriented and not merely a data-collection approach. The measurement program should be comprehensive and relevant to the business it serves. The measurement system should be able to accurately measure key metrics of end-user experience for understanding and improving our key services. There is tremendous complexity that must be carefully thought through, including several layers of technical architecture, varied organizational functions, service interdependencies and a disparate global user base.

All the measurements need to be consistent and measure the true end-user experience. In some cases, it might be impossible to measure true end-user experience. In those cases synthetic measurements can be deployed at key strategic locations in multiple countries. E.g. An test can be setup to run from a location at a regular interval to simulate user behavior and can measure the experience in terms of availability, performance, reliability, etc. Measurements must be repeatable and reproducible so that the

each team implementing the tools can expect to obtain results that have the same global meaning. Understanding the nature and properties of data collection and reporting is a core requirement to ensure that errors in measurement or faulty processes don't negatively impact the business or the group.

The end-user service quality expectation needs to get translated into service quality measurements. For example, the end user expectations that Hotmail should be fast needs to get translated into specific measurements for the top 3-5 scenarios in the top countries. The trend report should include the performance measurement along with the bytes download, files download, cache, expiry, tcp reuse, etc. These details are useful for the engineering team to analyze the root cause behind the problem and take action to fix them.

5.5 Instrumentation for Measurements

5.5.1 Measurement Method

All QoS measurements and data collection should be planned and their need must be planned to be in line with business goals. Based on the plan, measurement tools should be selected or built to collect the required data. The measurements & data collection should be automated to reduce cost and improve accuracy of the data. Measurements should represent all the major user-base and scenarios. The measurements should be minimally invasive with little to no impact or overhead to the user or service. For Example, the product can be instrumented to collect performance data (time taken to load the page across the globe) and report it back for analysis.

5.5.2 Passive Measurement – Data Collection/Reports

These are the typical measurements – where the data is collected, consolidated, and made available on the server for any analysis and reporting. This data is well suited to perform trend analysis and understand the issues in detail and prevent future problems. For Example performance and availability data collected from product third party for offline analysis and reporting.

5.5.3 Active Measurements – Alerts/Triggers

These are the active measurements – where thresholds and rules are set generating trigger points and are displayed on a dashboard and potentially emails are fired (e.g. if one of more servers go down or is slow). Appropriate alarms are set and people are notified. For Examples, Microsoft's System Center Operations Manager (SCOM), Microsoft's Operation Manager (MOM) and the tools developed by Microsoft that can be used to monitor service from server side. There are several third party companies like Keynote, Gomez, etc that provides active measurement options along with Triggers/Alerts/reporting and detail troubleshooting capabilities.

5.6 Data Analysis to find root cause

The key benefit of QoS focus is the continuous analysis of the service quality and improvement
For example, if we are seeing performance issues at the end-user and are unable to find the root cause to solve it, we need to at least inform the users of estimated time of completion in the UI and continue working on fixing the performance issue (e.g. when logging in to Live Mail, when it takes more than few seconds, the UI suggests the user to switch to old version of Hotmail for better performance).

In this phase, data collected in measure phase is analyzed using multiple tools (Cause-effect diagram, FMEA, regression analysis, statistical analysis, histogram, 5 whys, scatter plot, etc.) to Identify Sources of Variation and Determine Root Cause(s). Based on the outcome of analysis, team works together to validate the analysis and use that as a basis for coming up with improvement recommendations.

5.7 Making the improvements

This is the most important part of QoS forum. All the analysis done need to get converted into action to achieve the right set of results and sustain it. The team comes together and brainstorms on the root-cause of the issues and potential solutions and opportunities that will drive service quality improvement. Typically design of experiments (DoE) is used to develop potential solutions and validate it to see it will improve the process and minimize the impact of the root cause using pilot studies.

The list of potential improvements along with their pilot results analysis and return on investments are discussed with the stakeholders and then implemented. It is important to monitor the key process indicators (KPI) to ensure that there are no undue effects due to the improvements and the benefits are sustained – the next phase focuses on Control (also called as Sustain) phase.

5.8 Alerts/Triggers for Service Quality issues

Triggers will help the right people to get right information about the right service at the right time. If Quality of Service of a particular property is degrading, their owners need to be notified and root cause analysis need to be done. Most of the time, quality related information is not available immediately. Usually it needs an effort from a Project Manager (PM) or Analyst to analyze the information and then take action. Often, such decisions are delayed by two to three weeks (this sometimes can have a negative impact and we may lose end-users).

5.8.1 Key Metrics

For each product or service, the key metrics need to be identified. Different stakeholders will have different expectations for the metrics. Here are some examples of metrics for key stakeholders:

Marketing Manager - # of users registering every week, returning users, # of minutes on the homepage
Business Manager – Ads delivered, revenue, etc.
Quality Manager – Service Quality (Performance – average latency, % transactions less than 4 sec, Availability % uptime, Reliability – bug free transactions, etc.)
Operations – Capacity – Throughput (Transactions per seconds), Availability (% uptime)

5.8.2 Mechanism / Tool

Build a Dashboard / Scorecard, which can list down key metrics for the service/property and the Business Analyst/QoS Lead/Quality Program Manager can set the threshold for each of the metrics (Business Logic). Whenever one or more of the metrics crosses the threshold, a trigger can be set (email / SMS) and change the color of the alert to Orange. If this color remains in this state for a certain number of days, the tool will set the trigger to a higher level of management and the color will change to Red.

The tool should provide a Dashboard where the analyst or PM can see the health of their services at any moment of time (whatever is available latest). The tool will have a UI where the analyst and PMs can create new triggers and modify existing ones.

Alert(s) will be sent to the key stakeholders (e.g. regional PM, Operations, QoS team and to the related PM from the product team in US). The user can mark it as “investigating” and then work on finding the root cause. And when the issue is found and fixed – the status can be updated accordingly. We would need a tool that can get all the relevant data from other existing tools (in a timely manner, every 1 hr or 4 hr, etc.).

Example 1:
Business Scenario for absolute measurement of Quality of Service: If page load time (PLT) for mail.live.com users in Dublin goes above 15 seconds and remains that way for more than 48 hrs – set the first level of alert. If the situation does not improve within 5 days – set second level of alert.

Example 2:
Business Scenario for relative measurement of Quality of Service: If the hotmail transactions hitting DNS or TCP errors increases by more than 20% (consistently for more than 3 days) compared to the last 7 days average error rate - set the trigger. (Note: this will not catch gradual degradation of the service, a combination of absolute and relative might help).

Example 3:
Business Scenario for competitive measurement of Quality of Service: If the difference between hotmail and Google PLT (for past 30 days) becomes worse than the PLT of last month – set a trigger.

5.8.3 Threshold / Criteria

For all of these measurements – we need to finalize the formula and the criteria based on each of the metrics and measurements – e.g. 80% of the users. Average of all the values, 75 percentile, weighted average. BroadBand users only, value should only be considered if it is above threshold for more than 80% of the time, etc.

The criteria / threshold and triggers need to reviewed and updated as and when there is a major change in the business strategy – major service release and based on trending data. E.g. threshold for PLT was set to 15 seconds and for the past five months it has always been around 8-10 seconds. This needs to be reviewed and the trigger needs to be set at the right place (probably about 12 seconds) and similarly changes need to be made based on the changes in the application, etc.

5.8.4 Trigger Frequency

The intention of these triggers is not to fire on every change in the metrics (which might occur several times during the day due to several reasons – e.g. peak hours, maintenance hours, etc.). We should not have the same trigger fire every hour / every day.

Based on the severity of the service and the criteria defined, the business logic should be to monitor the trend of a logical time period, (e.g. average of past 7 days and compare it with earlier time period and then set a trigger. A time interval should be a moving window. But the interval should not be very big or very small. If we have too many false positives – people will lose faith and will not look at the triggers/alerts from this system.

Once somebody accepts the trigger and starts working on it, we should have a provision to mark it on the dashboard (bug #) and then reduce the frequency of the trigger (we don't want to bother people every day with the same trigger and at the same time we don't want people to drop the ball. So, the trigger frequency may need to change from about 3 days to 10 days).

6. Case Studies

Here are some of success stories of how Six Sigma based QoS approach helped us is performance testing and driving improvements in MSN/Live Search (Bing) and MSN Mobile UK and USA.

6.1 Live Search (Bing) Improvements

We implemented this approach of using QoS for MSN Search and drove significant improvement in performance and that resulted into higher customer satisfaction, increase in customer satisfaction score and higher ad-click thru rate.

Summary:
The migration from MSN Search to new "Windows Live" platform/code on Sep 14, 2006 added several files to the search code and the results page became heavier for the end-user. The page load time was impacted and it took 4.7 seconds to load (compared to 2.2 seconds before the migration). This was observed early October 2006 by the QoS and Performance management team and reported to Search Performance PM. After several weeks of discussion over the right tool, measurement methodology and approach, we decided to work together and solve the performance problem. Deep-dive analysis was done to understand the root-cause of the performance problems. With the strong partnership between Search, QoS and Performance team – we were able to solve the performance problems and make Live Search faster than Yahoo search and very close to Google search.

Here is the summary of the page load time for search results page – you can see the improvement achieved between Mar 2007 and Dec 2007.

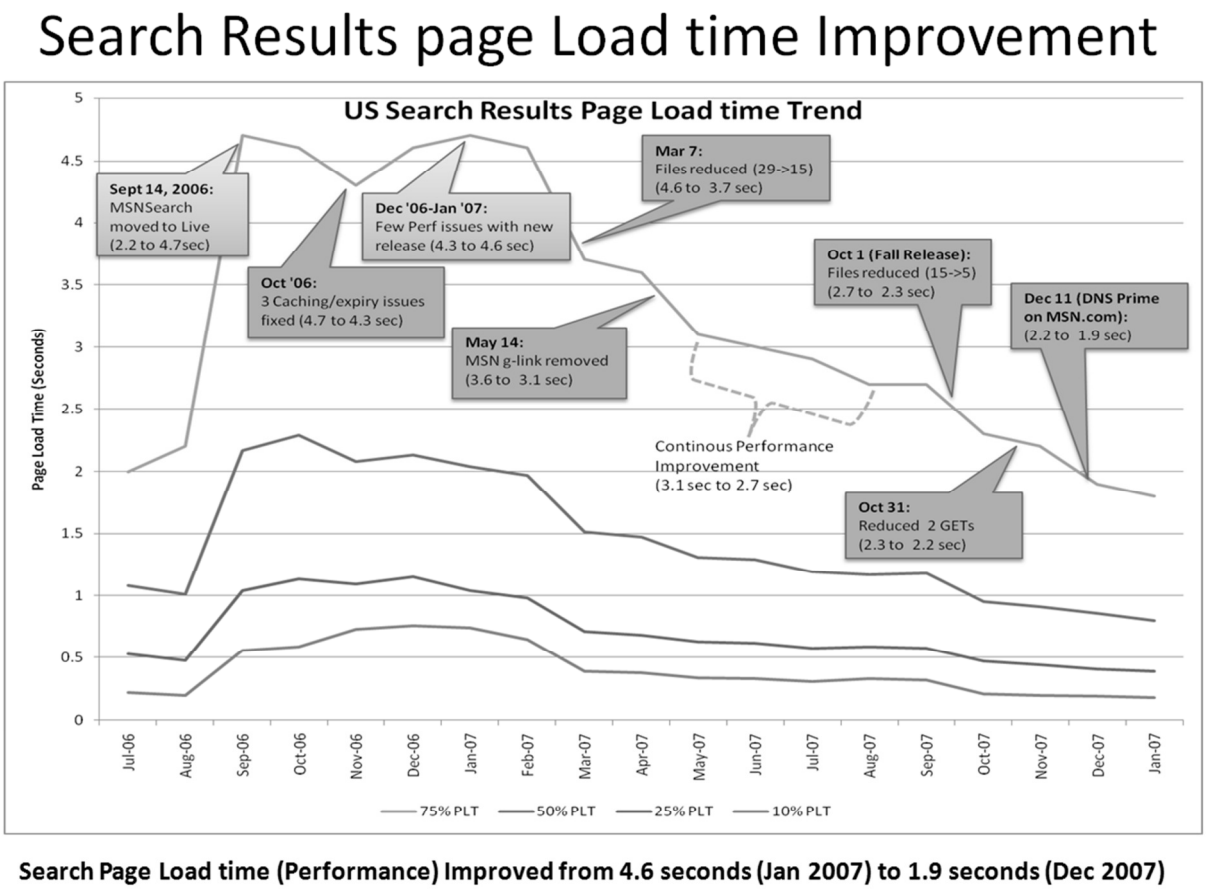


Figure 4: MSN/Live Search (Bing) improvements

6.2 MSN Mobile

Project goal: Capture In-Market Performance & Reliability measurements for key mobile services as they relate to the overall mobile service quality, analyse the data, publish the results to executives and drive improvements.

Observations: Mobile Measurements were done in US, UK and India for some of the services along with their competitors using Keynote's Mobile Monitoring technology (MDP – Mobile Device Perspective). We uncovered several issues just by doing a basic set of measurements of page load time. Using the QoS &

Six Sigma techniques we were able to identify issues and take necessary actions and put control in place to prevent such issues in future.

Every hour a measurement was taken on Mobile Device to measure time taken to load the page MSN and BBC. The observation was MSN was much slower than BBC and one of the changes in MSN made it slowdown further. For BBC portal it was observed that sometimes, the carrier takes different route which caused the page load time to fluctuate between 6 and 13 seconds.

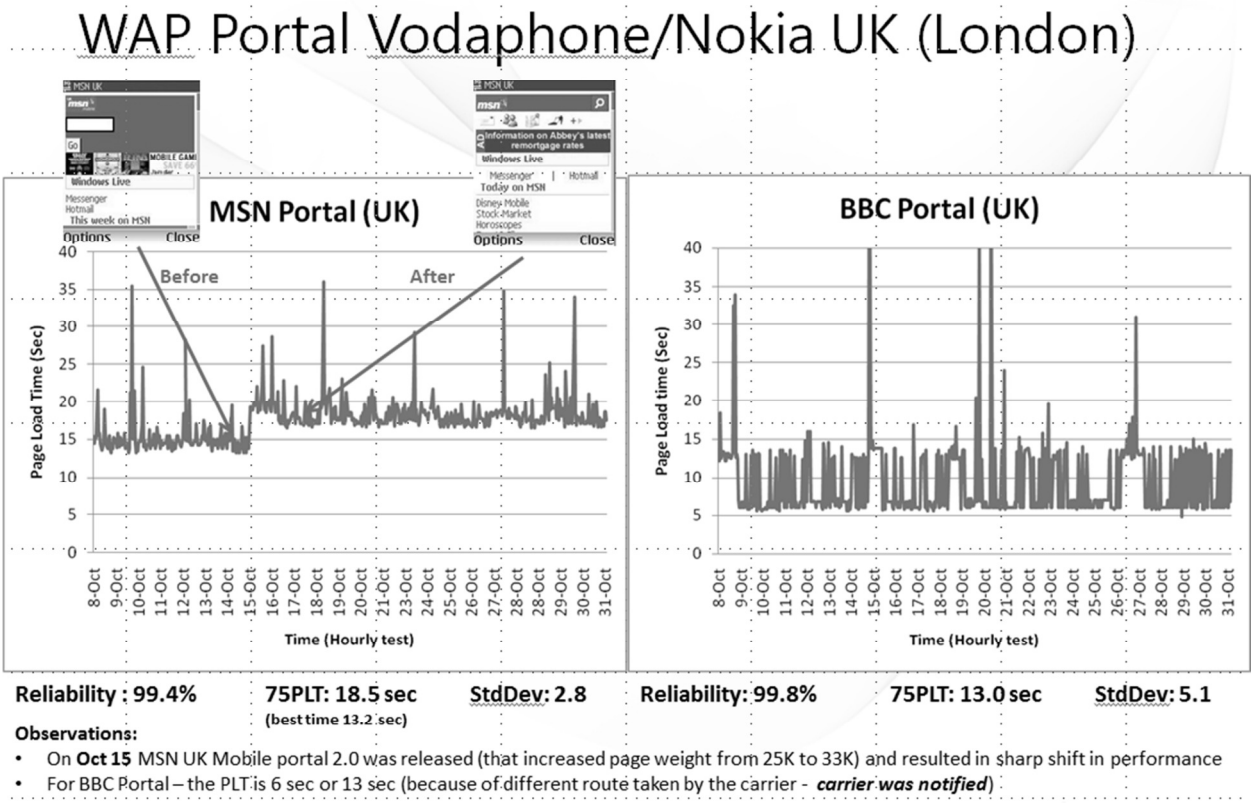


Figure 5: MSN UK Portal Performance Measurements

Similar measurement was done in New York on Moto Razr mobile phone for MSN US page. We observed the page load time was way too high (96-170 seconds), after analysis, we found that this was caused due to IE6 settings used by Moto Razr. We made some changes on the server side and fixed the issue and the performance improved to 20 seconds.

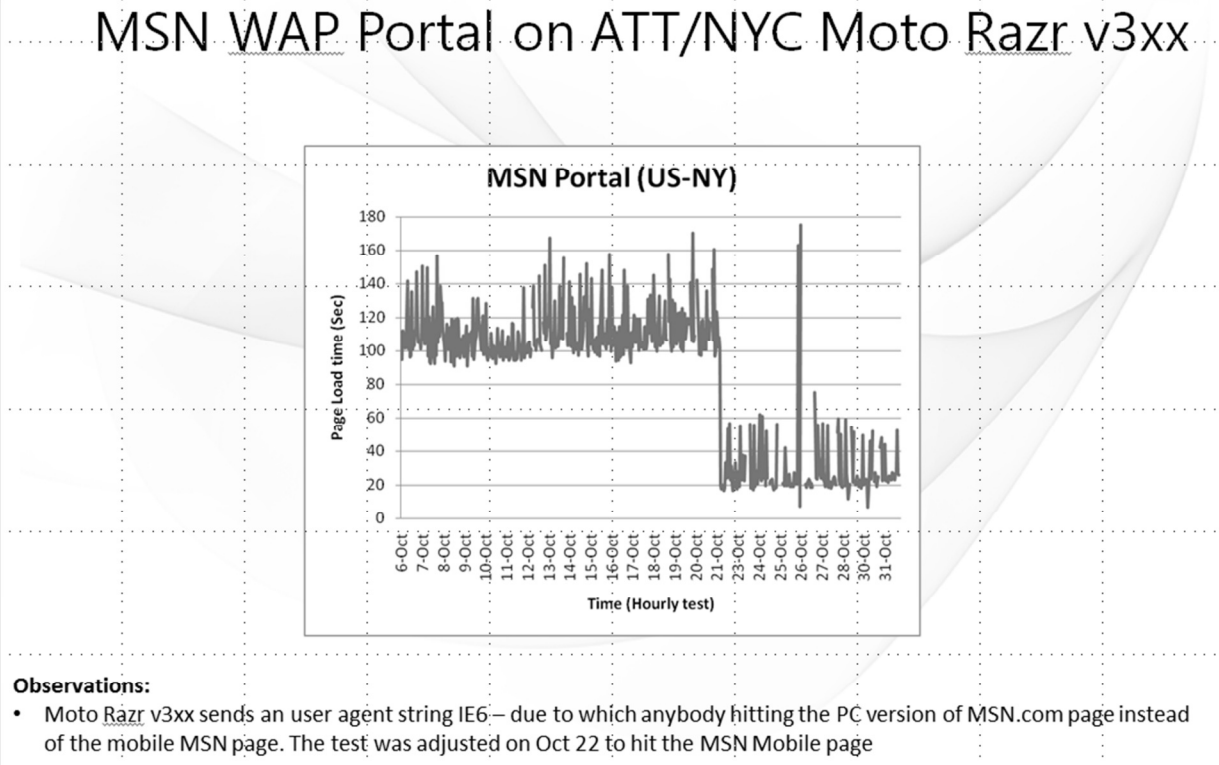


Figure 6: MSN US Portal on Moto Razr Performance Measurements

7. The Road Ahead

Similar QoS approach can be applied to several of our Software + Service offerings and mature/optimize service quality levels and delight our end-users.

High level map of work which can be arranged in a plan for next 3 month, 6 month, 12 month, etc.

1. Decide on key measurements that can represent overall service from end user perspective
2. Agree on consistent taxonomy of the key measurements, methodology and process
3. Invest in Infrastructure
    a. Build Standard Measurement engine
    b. Build back-end Storage to collect data and perform pre-processing
4. Measurements
    a. Understand each of the online properties scenarios and add transactional measurements to the system
    b. UI for teams to be able to modify measurements (based on their changes in code)
5. Reporting
    a. Common site with standard reports
    b. Tools to enable individuals & teams to build reports
    c. Create/customize alerts
    d. Tools to enable Analyst/Problem engineering specialists to access raw data for complex analysis

8. Conclusion

The paper presents an overview of how end-user experience with the service quality can be measured and how those measurements can be leveraged to continuously improve the quality of our Software + Service offering. This drives end-user satisfaction resulting into higher market share and revenue.

Delivering a high-quality product/service globally is not possible unless we have a good process in place and learn from our mistakes and are able to measure and manage it effectively. We have found Six Sigma methodology to be very helpful to measure and improve our metrics collection process with a goal of improving end-user satisfaction and driving business results.

With the use of Six Sigma methodology, we have seen the rigor in defining the right set of metrics for service quality, measurement focus, and providing a structured way of analysing improving and sustaining quality of the service. As soon as a team adopts the QoS Model, it becomes part of the culture and drive data-driven discussion and improvements.

While there are several ways to do measurements, it is key to identify the right measurement for the product and take into considering accuracy, cost, ease of use, etc.

9. References

[1] The Six Sigma Way: How GE, Motorola, and Other Top Companies are Honing Their Performance by Peter S. Pande, Robert P. Neuman, Roland R. Cavanagh

[2] Delivering Successful Projects With TSP And Six SIGMA: A Practical Guide To Implementing Team Software Process by Mukesh Jain

[3] Kano Model: http://en.wikipedia.org/wiki/Kano_model

[4] Six Sigma DMAIC (Define, Measure, Analyse, Improve, Control) http://en.wikipedia.org/wiki/Six_Sigma

Testing the Mobile Application's Performance: Case Study on Windows Mobile Devices

Rama Krishna Pagadala

Unified Communications Group
Microsoft Corporation

Email: Prk.Reddy@microsoft.com

Abstract

Mobile software usage is growing quickly, and an expanding number of consumers are expecting a consistent experience when moving from desktop PC to mobile devices and back again. One of the key factors enabling wide deployment and adoption of mobile applications is careful usage of system resources. High quality mobile applications must consume fewer system resources such as CPU, Memory, and most importantly the Battery life. It is crucial that design for mobile applications considers frugal and optimal resource usage as an essential aspect of application design (direct porting of desktop applications to mobile platforms is most often a recipe for failure). Knowing and understanding the factors that influence performance on mobile devices, is extremely beneficial in order to plan a successful testing approach. Equally challenging is the task of measuring application performance and reporting the performance data in a useful and actionable manner.

In this paper, we will present a case study on performance testing of the Microsoft Office Communicator Mobile application for Windows Mobile 6.x phones. We will discuss our approach to performance testing and the challenges involved in measuring various metrics. We will also provide details on the performance metrics, tools used to gather and present this data, and how we have used this data to make important decisions throughout the product cycle. Some architectural improvements have contributed to improve the application's performance and better usage of system resources. For example, the battery life has improved by more than 300% in the 2007 R2 version of Office Communicator Mobile.

Biography

Rama Krishna has been working at Microsoft as a Software Design Engineer in Test for over five years. He has been extensively involved in testing the Office Communicator Mobile application on various mobile devices including Windows Mobile 6.x series devices. He has also designed and implemented a tool to test the DHCP server's performance. Prior to joining Microsoft, he worked as a Software Development Engineer at Cisco Systems for four years. He earned a Master's degree in Computer Science in 2001.

1. Introduction

One of the biggest challenges the Mobile Information Workers (MIW) – also referred to as Business Power Users (BPU) in this paper – are facing today with the increased use of devices is the battery life of the mobile devices. As the mobile applications base rapidly increases on the small form factor devices without any significant improvements made to the overall battery life, it is critical for the applications to preserve the battery life.

Next big challenge is the inherent flakiness in the cellular data networks which demands the mobile applications to be robust with the network loss. And, pay-per-use data plans in some countries emphasize the need for reduced network usage on the slow speed cellular data networks. The consumption of both the battery life and the network bandwidth matters more for the applications like Communicator Mobile that keep the users always connected.

The real challenge for the development and test teams is to measuring the performance metrics accurately that reflect the end user's environment. This paper presents the test strategies used to measure the performance metrics, specifically the battery life and the network bandwidth, that the mobile users will notice when they use the Microsoft® Office Communicator Mobile application on their Windows® Mobile (6.x) devices. We will also discuss the tools used to measure these metrics, and how these metrics have been reported to the product team and to the management.

Based on our test results, there were several improvements made to the Communicator Mobile 2007 R2 client to reduce the application's impact on the mobile phone's battery-life and to reduce its bandwidth utilization. Some efforts were made to reduce the overall CPU and Memory utilization which are outside of this paper's scope.

The goal of this paper is to aid the Development and Test managers aiming to improve their application's performance by providing the test strategies, tools, and the important metrics to focus on. Test teams defining the performance testing strategy would greatly benefit from this case study as well. Majority of the tools used in our testing are publicly available.

Section 2 starts with *what* is being measured (i.e. performance metrics), section 3 covers *how* we have measured them (i.e. testing strategies), followed by what are the *key scenarios* and *targeted users* for which these metrics are being measured are discussed in section 4 and 5, and finally section 6 and 7 cover how the results are *reported* and what *tools* are used to measure and report these numbers. Section 8 provides conclusions and next steps for the readers interested in using performance approach discussed in this paper.

First, let us take a look at various terms/acronyms used throughout this paper:

Definitions/Acronyms	
CoMo	Microsoft Office Communicator for Windows Mobile devices
OC	Microsoft Office Communicator for Windows PCs
Mobile Phone	Any Windows Mobile device running 6.0/6.1/6.5 OS version
Pocket PC (PPC)	Touch phone with or without qwerty keyboard – loaded with Windows Mobile Professional 6.0/6.1/6.5 OS
Smart Phone (SP)	Phone with a qwerty keyboard, but no touch screen – loaded with Windows Mobile Standard 6.0/6.1/6.5 OS [Refer http://www.microsoft.com/canada/windowsmobile/wm07/about/faq.mspx#2 to understand the difference between a Pocket PC and a Smartphone]
Business Power User (BPU)	Refers to the Mobile Information Worker (MIW) that uses the Windows Mobile device for the day-to-day activities like phone calls (personal & work), Email/Calendar, and Web browsing etc.
Days of Use (DoU)	Number of days mobile device can be used without recharging the battery

2.5G and 3G	Cellular data network types with different characteristics like upload/download speeds etc.

2. Metrics

In this section, we will talk about the key performance metrics measured. Please note that these metrics are collected from multiple test runs to make sure that the variation across the runs is adjusted. Refer to section 4 for the *user profiles* referenced in this section.

2.1 Battery (in mAH)

Battery consumption can be measured using the Power Monitor tool (refer to section 7.1) which measures the ***current*** i.e. rate of discharge in mA. The full size battery charge is usually represented in mAH and hence we have reported our measurements in mAH as well. The formula for calculating the total discharge as measured in mAH within a specified time is:
Total Discharge (mAH) = Rate of Discharge (mA) * Total Duration (sec) * 1 (Hr) / 3600 (sec)

For example, if the rate of discharge during one hour steady state (i.e. no user activity) is measured as 100 mA, then **Total discharge in one hour** = 100 mA * 3600 (sec) * 1 (Hr)/ 3600 (sec) = 100 mAH
For a battery with 1000 mAH charge it amounts to a discharge of 10% by the end of the hour or in other words the battery will take 10 hours to discharge completely under steady state (i.e. linear discharge).

Power consumption is measured for different scenarios. It is the battery consumption at the device level and is not feasible to measure it for a particular application. Based on the user profile (refer to section 4), the total power consumption is being calculated.

One caveat though – the expected variation for the battery measurements is up to 10% (it can as well be up to 20% due to dramatic weather changes, device density in the lab etc.) and thus the battery results should be used only for the planning purposes.

2.2 Bandwidth (in KB)

The bandwidth consumption is measured and reported in Kilo Bytes. For CoMo and OC power users, it is the total bandwidth consumed by both ActiveSync (Email) and the CoMo application. And, it is just the bandwidth consumption of ActiveSync application only for the Email power user.

This metric refers to the network bandwidth consumption for various activities. It helps the user understand the *total bandwidth consumption per day*. It becomes critical for the users with the pay-per-use data connections (data plans have been expensive in Europe and Asia). A lot of improvements were made in 2007 R2 version of CoMo to minimize the network activity thereby reducing the total bandwidth consumption (per day) by CoMo application.

2.3 Days of Use (DoU)

Unlike Battery and Bandwidth measurements, Days of Use (DoU) is not a metric that is measured using any tool, but it is being calculated with respect to a baseline battery so that different power users will have a sense of how long the battery would last for them. The formula used to calculate this DoU is:
DoU (days) = (Total battery power i.e. mAH consumed in 24 hours) / 1000 (mAH)

Note: Total battery consumption is extrapolated for the 24-hour time period (refer to section 4.2). For the end users (viz. mobile application customers), this metric is relatively more interesting than the absolute battery consumption in mAH (section 2.1).

Days of Use for different power users is being calculated with the reference battery of **1000** mAH. The same reference battery is used for both the Pocket PC and the Smart Phone which would allow us to compare the trends across the device flavors. However, in reality, different devices come with varying battery capacity and thus the Days of Use should be calculated with the actual battery capacity of the device. For example, the Pocket PC device used in our testing (ATT Tilt) comes with 1350 mAH battery.

Also, the Days of Use calculations take the full battery capacity into consideration. It would mean the battery will drain completely at the end of *Days of Use* time period. However, please note that, the Windows Mobile OS and some OEMs enforce the minimum battery levels for the devices to continue to be functional which would vary from 10-20% of the remaining battery. So the Days of Use will always be lower than what is reported. Also, the battery drains much faster when it reaches certain level (around 20%) which could impact the DoU duration. Hence the DoU reported is based on the linear battery model.

3. Test strategies

Three kinds of performance testing approaches are used:

1. Days of Use – aggregated numbers for the typical usage of the device
2. Regression testing – granular numbers for the individual scenarios
3. Internal self-hosting – anecdotal numbers based on product team's usage of the application

We will not discuss in detail about the anecdotal numbers which were collected from different sources like server logs/SQM counters, device side logs, and user inputs. We have compared these numbers with our lab results to get a sense of how reflective our measurements are to the end-users.

3.1 Days of Use (DoU) Testing

Note: do not get confused with the *DoU metric* described in section 2.3 with the *DoU Testing* as a testing approach, even though both are inter-related.

The primary goal of the Days of Use (DoU) testing is to find out how long the battery will last and how much bandwidth would be consumed for different Business Power Users (BPU). To achieve this goal, various user profiles are defined to group various mobile information workers. The idea here is to measure the battery and bandwidth consumption for these user groups on different devices that are connected over different networks (2.5G and 3G). One can interpret the data to understand:

- How much battery is consumed for an user type using a particular device (PPC vs. SP)
- How much bandwidth is consumed over a particular network (3G vs. 2.5G)
- How much more battery and bandwidth will be consumed by CoMo as an application
- How the network type (3G vs. 2.5G) impacts these metrics
- How the device flavor (Pocket PC vs. Smart Phone) impacts these metrics

We will discuss in detail about the user profiles, time profile, and the scenarios used to measure the metrics in the sections 4 and 5.

3.2 Regression testing

The goal of the regression testing is to find out if the performance has been improved (or degraded) from an older build to the latest. These are the numbers application developers would be interested in as they need to investigate the root cause for the change in numbers. First thing they look at would be the code changes that went into the latest build and see if they impact the performance.

For CoMo, we have defined a set of granular scenarios to measure and report the numbers. These are low level scenarios that map to specific user actions. Find below some sample scenarios used to measure various metrics:

Scenario
First time sign in from CoMo with an user that has 100 contacts
Re-Sign in with the same user (100 contacts)
Sign out with the same user (100 contacts)
Update the user status (Presence) from CoMo
Keep-alive traffic only (100 contacts, 60 minutes) i.e. CoMo stays signed in with no user actions
Launch a contact card from contact list
Scroll down by 10 contacts (and wait for the presence update)
CoMo user establishes IM channel and sends IM
Send a message (in an active IM conversation) from CoMo

4. User profiles

User profile is nothing but an ordered set of scenarios/actions that reflect the typical work day of a particular user a.k.a. targeted persona. It consists of set of activities along with their frequency that would describe the typical day of the user. Defining a user profile is the first step in developing any application/service to describe the target audience and then design the features for that audience. We have applied the same notion to our performance testing as well.

We have defined three user profiles to represent three large segments of the mobile information workers (or business power users). One of them does not use Communicator Mobile and is considered as a reference user. And the other two power users use CoMo and the results are compared against that of reference user to find out the delta (i.e. resources consumed solely by CoMo application). Also, there are multiple scenarios covered in the test matrix to reflect on the power users' usage of the mobile devices in their day-to-day life.

In this section, we have presented the activities considered for the CoMo power user and how these activities are being spread across in 24 hour time period. Section 5 will cover the scenarios for all the three user profiles.

4.1 CoMo Power User profile

We have defined the user profile for the CoMo power user as shown below. Note: it is not exhaustive by any means and is provided as an example.

Type	Details
Sign-In / Sign-Out	• 1 Sign-In/Sign-Out every 2 hours due to network outage, or phone in use (for 2.5G) etc.
Contacts	• 80 members in contacts list • 1 scroll contact list by 10 users

Instant Messaging	• 1 IM session every one hour with 12 IM sessions per day
	  ○ 6 are incoming IM invites (2 receive IM and 1 send IM)
	  ○ 6 are initiated by the user (2 send IM and 1 receive IM)

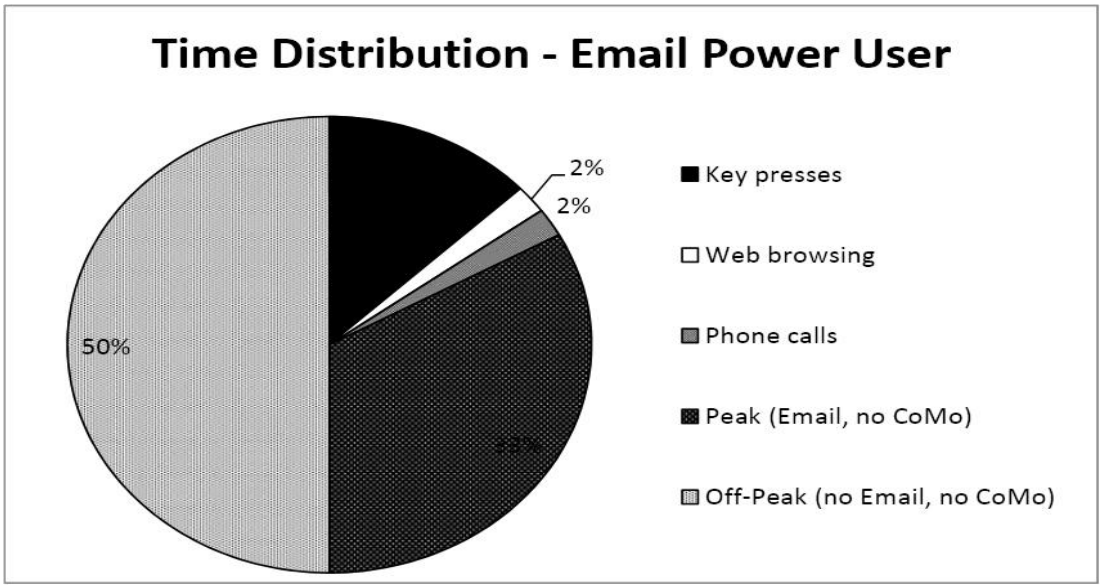
	○ Each IM on an average has a 20 characters
Phone Calls	• 12 phone calls per day with 1 phone call per hour ○ 1 incoming call every two hours ○ 1 outgoing call every two hours • 2.5 minutes per call (talk time)
Presence	• 6 Self-presence updates/hour due to user activity on the device every 20 min • 4 Presence updates/contact/hour (not to exceed 240 updates in an hour) • 50% of the contacts change presence at the hour

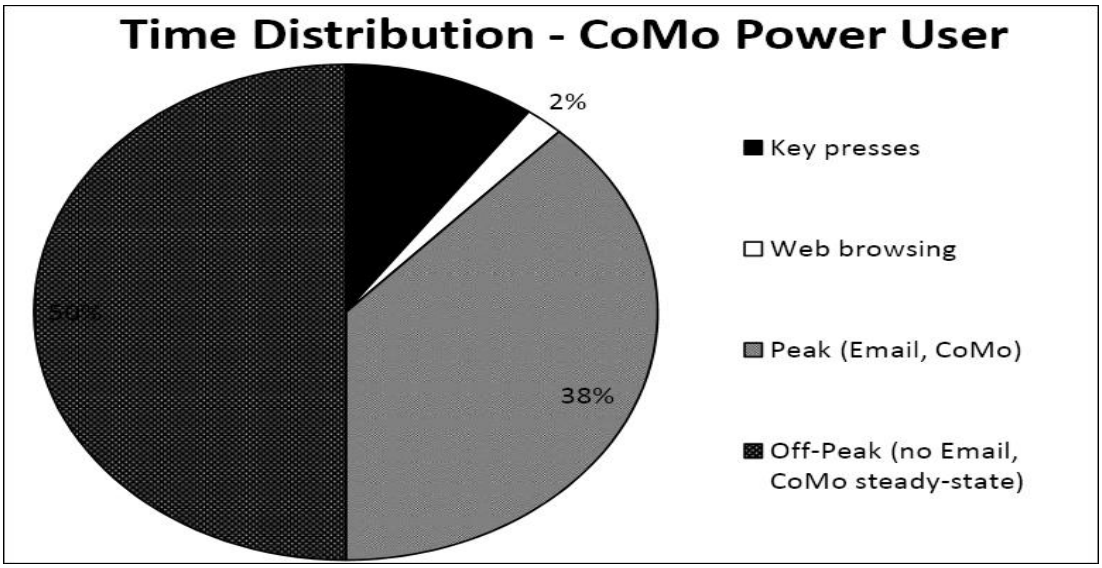
4.2 Time profile

We have defined the time length, in hours, for each scenario and profiled the 24 hour day for different power users as shown below:

Scenario	Hours per Day		
	Email PU (Email + CoMo)	**CoMo PU (Email + CoMo)**	**OC PU (Email + CoMo + OC)**
Key Presses	3 hours	2.5 hours	2 hours
Web browsing	30 minutes	30 minutes	30 minutes
Phone calls	30 minutes	N/A (see CoMo/OC activities)	
Email activities	8 hours		
Email - Off Peak	12 hours	N/A (see CoMo Off Peak)	
CoMo - Off Peak	N/A	12 hours	12 hours
CoMo activities (with background Email)	N/A	9 hours	4.5 hours
CoMo and OC activities (with background Email)		N/A	5 hours
Total (hours)	**24**	**24**	**24**

Key presses are to emulate the users' usage of the device keypad i.e. while sending an email, setting a note in CoMo etc. Email activities above refer to the background email using Active Sync Push Email. Here is the pictorial view of the time profile for Email PU (reference user) and CoMo user.





Please note that the activities like Web browsing, Phone calls and the Email are all considered as part of the Peak time for the Email PU.

5. Top Scenarios

Find below the list of scenarios that comprise the overall testing and some of them are common to all the power users and some differ for different users.

5.1 Definitions and common scenarios

- **Off-peak time:** This is the time period that the user does nothing and only the minimal background activity is expected during this time. And the activity varies for different PUs as described below.

- **Peak time:** This is the time when the user is using the device extensively. And the activity varies for different PUs as described below.

- **Peak Email:** ActiveSync on device is configured to sync the emails in real-time. And the user receives the emails at the frequency of ONE email every 10 minutes during the peak times. Each email will be of size 1 KB (HTML formatted) and we have setup the incoming emails only.

- **Phone calls:** User initiates and receives the cellular calls on mobile. The frequency of the phone calls (either incoming or the outgoing) is one call per hour during the peak times.

- **Web browsing:** User visits different web sites that contain the following content.
  - ✓ Text-only
  - ✓ Images-only
  - ✓ Both Text and the Images

  Pocket IE was used for this purpose and the continuous browsing was considered during the peak times only. The backlight is set to be ON throughout the test.

- **Key presses:** This is not a scenario, but it is the time when the user is using the device for all the above listed activities. For example, there are a few key presses involved in making/receiving the phone calls (which are not accounted for in the **Phone calls on mobile** scenario).

5.2 Email Power User

In this section you will find the scenarios that are specific to this user and/or that differ from other power users.

- **Off-peak time:** The only activity during this time is the ActiveSync keep alive messages with the email server. In our testing, even though no emails are sent to the user during this time, the ActiveSync constantly checks with the server for the new emails (or it simply maintains the active connection with the server and server pushes the emails down?). It would consume some battery and the bandwidth both of which are found to be marginal.

- **Phone calls on mobile:** User initiates and receives the cellular calls using the Phone **Dialer** interface on WM.

5.3 Scenarios common to CoMo and OC Power users

In this section you will find the scenarios that are specific to this user and/or that differ from EMail PU.

- **Off-peak time:** In addition to what is being described for the Email PU off-peak time in section 5.2, the background activity of the CoMo application is also considered here. It would include the keep-alive messages with the OCS server so as to prevent the registration from expiring (which would other cause the user to sign out).

- **Phone calls on mobile:** CoMo user initiates and receives the cellular calls using the Call-via-work feature available in CoMo 2007 R2.

5.4 Key Test Areas

In summary, the performance metrics are measured for various combinations of the below areas.

User profiles:

- Email PU

- CoMo PU
- OC PU

Device flavors:

- Pocket PC (touch)
- Smart Phone (non-touch)

Cellular networks:

- 3G (UMTS)
- 2.5G (GPRS)

6. Test Results Reporting

Test results have been reported as absolute numbers as well as aggregated numbers that the end-users would notice (they would care). The absolute numbers are useful for the engineering teams to find out the areas of improvement and the aggregated numbers are useful to get a sense of what the end-users will experience. Some of the architectural improvements to enhance the application's performance have been published in Office Communicator Blog.

For the benefit of the reader, we have provided some sample analysis and graphical representation of the data in this section.

6.1 Absolute numbers

We have presented the results in different formats. For each power user profile, the absolute numbers for a particular device are being analyzed. Also, the numbers are being compared with other users or other device/network for the same user. Sample data along with a brief analysis is provided below for Pocket PC over 3G for 2007 R2 version of CoMo.

Email Power User

Scenario	Power (mAH)	Bandwidth (KB)
Key Presses	316.08	0.00
Web Browsing	113.92	2000.00
Cost of Phone Calls	149.70	0.00
Peak Usage (Email + no CoMo)	76.64	99.20
Off Peak (no Email, no CoMo)	70.56	38.40
Total	**726.90**	**2137.60**
Days/Hours of use	**1.38 Days / 33.02 Hours**	

Based on the above results, it seems the phone would last around 33 hours for the Email PU and the majority of the power consumption comes from key presses and the phone calls. However, the bandwidth was mostly consumed by the web browsing.

CoMo Power User

Scenario	Power (mAH)	Bandwidth (KB)
KeyPresses	263.40	0.00

Web Browsing	113.92	2000.00
Peak Usage (Email + CoMo)	519.98	850.50
Off Peak (no Email, CoMo running in the background)	112.08	214.80
Total	**928.46**	**3065.30**
Days/Hours of use	**25.85 Hours = 1.08 Days**	

For the CoMo PU, the majority of the power consumption comes from the CoMo activities followed by the key presses. The later is partly due to the amount of time spent by this user for the key presses. The major stake holder of the bandwidth consumption still seems to the web browsing though.

OC Power User

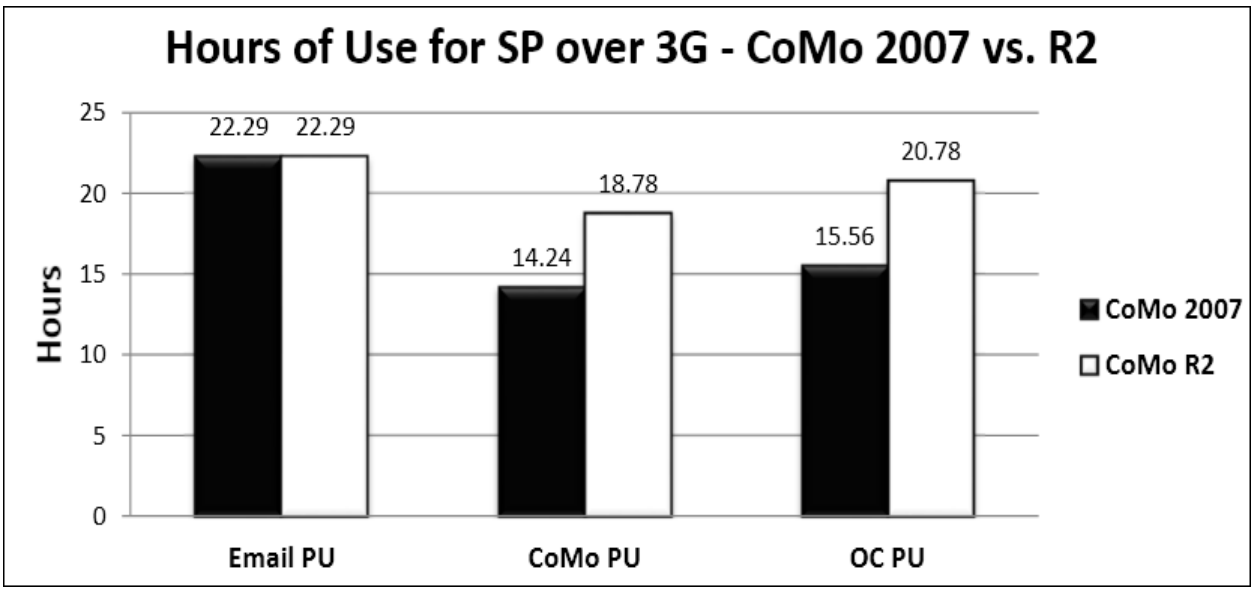
Scenario	Power (mAH)	Bandwidth (KB)
KeyPresses	210.72	0.00
Web Browsing	113.92	2000.00
Peak Usage (Email + CoMo)	219.53	425.25
Peak Usage (Email + CoMo + OC)	185.50	324.50
Off Peak (no Email, CoMo running in the background)	112.08	214.80
Total	**841.75**	**2964.55**
Days/Hours of use	**28.51 Hours = 1.19 Days**	

The power consumption of this user seems more interesting as it is more scattered for different scnearios. Please note that more than 20% of the battery consumption was noticed when the user was actually using the OC (MPOP) endpoint.

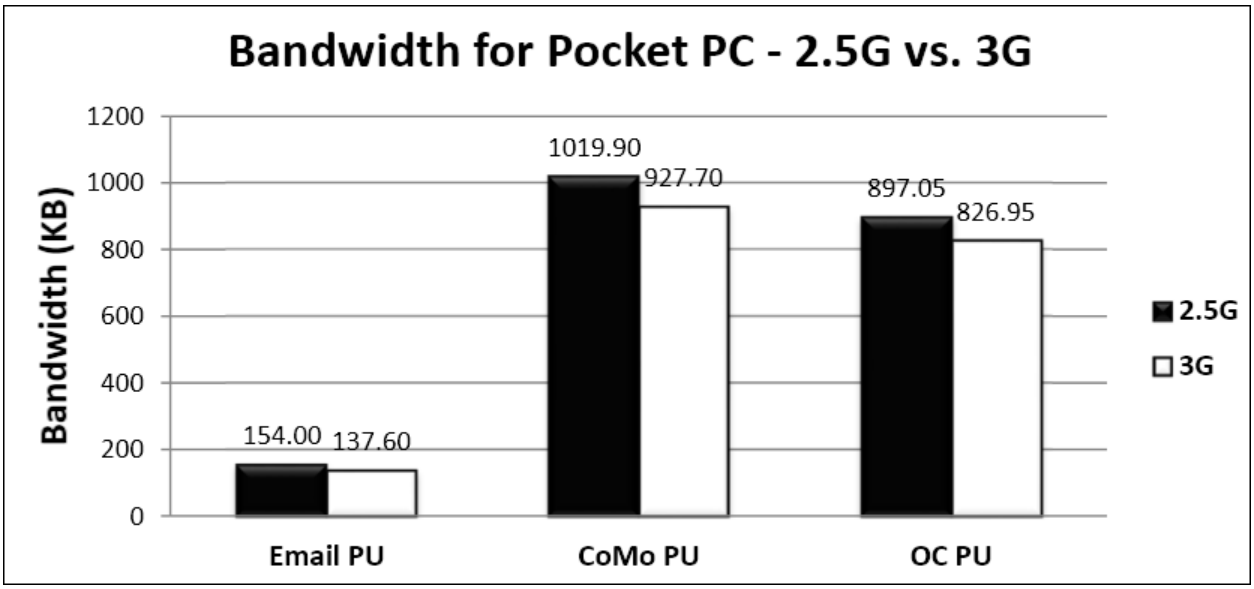
6.2 Comparison of different power users

Sample analysis is provided below for Smart Phone's power consumption and Pocket PC's bandwidth consumption.

Our test results indicate that the overall power consumption of the Email power user is significantly lower than other power users which will tranlate into longer Days of Use (lasts longer). Results also indicate that power consumption on a Pocket PC is relatively less than a Smart Phone and it was due to the differences in the power management on these devices. And the power consumption is relatively higher for 3G network compared to 2.5G network. However, the bandwidth consumption is slightly higher for 2.5G network. The reason for the higher power consumption by 3G network is due to the fact that it keeps the radio in high power mode for around 10 seconds after the data transmission is complete.



The bandwidth consumption for the web browsing would greatly vary depending on the browsing needs (text only, images, videos etc.). Our results hinted that an additional 930 KB (< 1 MB) is consumed by CoMo application if the CoMo was used extensively (CoMo PU) and it would be around 830 KB additional bandwidth for the OC PU. The results varied significantly across the device flavors, moderately high across the networks and significantly higher for the CoMo client versions, 2007 and R2.



7. Tools

In addition to our home grown tools used to measure all the metrics mentioned above, we have also used some of the external tools to validate some of the measurements. We will not go deep into how to setup/use these tools in this paper and instead, provide some pointers to these tools and encourage you to explore them (and other related tools) further.

7.1 Power Monitor

Purpose: To measure the power consumption of the device

Source: http://www.msoon.com/LabEquipment/PowerMonitor/index.cfm

Version: 4.0.0

Setup: Connect the PowerMon hardware to a Windows PC using the USB cable and load the drivers to manage the device, including reading the samples collected by the tool. Device under test should draw the power from PowerMon so that PowerMon can measure the power consumption which can then be read on a PC

7.2 Spb Wireless Monitor

Purpose: To measure the bandwidth consumption of different applications using the GPRS network

Source: http://www.spbsoftwarehouse.com/products/gprsmonitor/?en

Version: 3.0.1, Release date: Dec 11, 2008

Setup: Install the software on device and use the appropriate settings (bind it to the GPRS network)

8. Conclusions and Next Steps

Performance testing is very challenging, particularly on mobile devices. The biggest challenge is the unavailability of the tools for these relatively new and less powerful devices. It gets worse with the proprietary platforms that are still evolving. Also the application's usage varies significantly by different mobile users.

Our first step in testing the performance of Office Communicator Mobile (CoMo) was defining the user profiles that accurately reflect the most common uses of the application:

- Users who are on the go and use CoMo for the most part, and
- Users that use CoMo in conjunction with Office Communicator on the desktop

Key step towards measuring the performance of applications like CoMo is defining the baseline user who does not use CoMo, but use other applications like Email and Phone calls. The next step is defining the metrics to be measured along with some improvements targeted for the release. The challenge is finding the right set of tools which will not only measure the metrics of interest, but they do so accurately. You may end up building some in-house tools to process the data gathered by 3rd party tools.

We have measured various *metrics* using Pocket PC and Smart Phone *devices* over popular cellular *data networks* and have provided the comparative results for *different power users*. The goal here is not necessarily to provide any recommendations about particular device models or particular network-to-use. Instead, these results should help the power users understand the implications of various activities (Email, Instant messaging etc.) on critical resources like battery and the bandwidth. They will also help the users understand how the resource consumption varies among the device models and the network types.

Key learning here is that performance data needs to be gathered several times over the time to better manage the outliers and also to assert the earlier measurements. As with any lab results, customers may notice slightly different numbers than those measured in the lab environment. It is also a good idea to perform a reality check anecdotally.

Some of the testing strategies came from Live Messenger Mobile team and the tools were leveraged from Windows Mobile development/testing teams. Most of these test strategies along with some of the tools are being used for testing the CoMo performance on 3rd party platforms like iPhone.

References

Document	Location
CoMo 2007 R2 Product Documentation	http://www.microsoft.com/downloads/details.aspx?familyid=59306902-B3E1-42CE-B179-90AFAD415F2A&displaylang=en
OCS 2007 R2 Product Documentation	http://www.microsoft.com/communicationsserver/en/us/product-documentation.aspx
Monsoon Solutions Power Monitor	http://www.msoon.com/LabEquipment/PowerMonitor/index.cfm
SPB Wireless Monitor	http://www.spbsoftwarehouse.com/products/gprsmonitor/?en
AT&T Tilt specifications	http://www.htc.com/us/SupportDownload.aspx?p_id=67&cat=2&dl_id=94
Black Jack II specifications	http://www.wireless.att.com/businesscenter/blackjack2/

Increase QA Team Efficiency by Utilizing Offshore Resources

Hao Zhao Expedia, Inc.

haozhao@expedia.com

Introduction

Following the telecommunication and Internet expansion of the late 1990s, the growth of IT services offshoring is linked to the availability of large amounts of reliable and affordable communication infrastructure. Especially with the digitization of many services, it is possible to shift the actual production location of services to low cost locations in a manner theoretically transparent to end-users. The economic logic of offshoring is to reduce costs, which is widely practiced and well understood (Henley, 2006). Therefore, employment or utilization of offshore has become more and more popular only only because it is considered to be a standard of software development strategy, but also because it is viewed as a key component on gaining advantage over other competitors. According to Lewin & Heilmen (2007), software companies, among other kind of companies, are early adopters of offshoring practices. Manning, Massini & Lewin (2008) reported that the employment of offshoring in the IT field increased from 5% in 1997 to 50% in 2007.

However, due to several reasons, offshore projects are confronting some issues, such as: unable to deliver the end result; fail to meet expectations; or even worse, doom to fail from the start. Like any innovations, issues are naturally along with development. With proper management and planning, these issues would be resolved and offshoring would become a key asset in IT teams. In this paper, I will present some of the key issues in the offshore planning and management at my work, share my experiences in the offshore management, and finally some lessons learned from past successful and failed projects.

My Experience of Managing QA Teams

I have been in the QA field for more than 12 years and managing offshore for more than three years. My team ranges from mid size (12 members, five onshore and seven offshore) to small (five, one onshore and four offshore). Onshore team are domestic (currently working with development team in three locations – Seattle, San Francisco, and Boston), offshore members are located in Asia (China, India, Philippine, etc). I have managed projects range from custom interface to internal tools, include standalong desktop app, web application, performance testing, ect.

Key Issues of Offshoring QA Teams

Having an offshore component in IT teams adds more complexity to the team and the project management. Major issues with offshore team can be considered as communication difficulty, which leads to mis-understanding and different interpretations. A second issue is that different expectations were assumed between parties. Another challenge is assess the strength and weakness of each team component. These are major issues I confronted at my management of a QA team and I will discuss more specifically in the following section about these issues.

1) Communication

One of the key hurdles is to be unable to communicate clearly with the offshore team. This is mostly due to language barrier and culture differences (esp. in the Asian region) between the two locations. The difficulty appears several symptoms: 1) different geo-location; 2) language barrier; and 3) culture differences. We all understand the difficulties of communicating with team members in a different geo-location. Most engineers in offshore (I assume mostly in China and India) have basic understanding of English language, their reading/writing skills (5/10 ?) are much better than speaking skills (3/10 ?). So it's always better to use email as the standard way to communicate. Moreover, Asian people don't speak more than Americans because the Asian culture respects listening unless are asked to speak, especially to the people who hold a higher status. My offshore team members are less willing to speak up (this is especially true in offshore team in China). To the degree where it will cause miss-understanding and miss-communication. I found that either hiring a onshore manager who speaks the target language or having an offshore lead who has working experiences in US will greatly reduce these risk.

2) Clear entry and exit queries

Because of geo-separations and poorly established process (which most likely not due to its large overhead), it's very difficult to share common goals and expectations. After working with offshore for a while, I was aware of the amount of non-official communication I took for granted by having my team sitting next to me; assign projects based on the evaluation; and at the same time, consider potential risk of those assignment to not only the QA team, but also to the project overall. Establishing clear entry and exit queries is always true to onshore teams, but especially important when you manage offshore teams. Clear entry and exit queries for offshore tests can act as unit tests for a development team. It's a self-evaluation toolkit before handing over the final result to onshore team for review. These queries must be measurable and agreed by both sides. Ideally the requirements will comes from test managers/leads in both onshore and offshore teams, this not only give offshore team a stake in project ownership, but also makes them feel more involved in overall testing strategies.

3) Improve test process and its transparency

One of the challenges of managing offshore teams is how to assess the strength and weakness of each team component and how to evaluate their project productions. Do the defect templates in the defect tracking system provide enough information for triage and investigation? Do I have a configuration management process in place? These questions might be passed by without notice when you only have a local/onshore team, but the issue would get magnified and potential impact when you deal with an offshore team.

Lesson Learned from My Experiences

I will discuss some management strategies from my experiences of working with offshore teams.

1) Hire a QA manager who has offshore management experience

This person has to be a great QA manager with hand-on experiences. He/she can not only manage people and projects, but also can roll up sleeve and show other team members how to do certain things. Also previous experience is very important because only through real life experience you can understand the pain of initial adoption of offshore and the joy when the team matures

2) Constant communicate

Use any necessary means to communicate between team managers and team members, as well between team members. Ask the offshore team to send out a meeting summary with clear action items and schedule. Have team member keep a journal (tweet is a great tool here) to update progress. Create an offshore team wiki page so that people can upload best practice and kb articles.

3) Choose appropriate projects (or components of projects) for offshore

Projects that have clear goals and precise execution path are great fits for offshore. Any kind of API testing is a great example of offshore projects. With proper documentations, offshore teams would need very little supervision and can produce high quality automation tests. There are certain projects that are not a good fit to be offshored. For example, projects that either requires constant communication or don’t have a clear specification or designs.

4) Stike a balance between offshore and onshore

Onshore or local teams provide risk-relief for the team organization. There are several risk involved when you offshore: a) communication is difficult, and b) potential loss of domain knowledge. Having a well balanced onshore-offshore team ratio is a great way to manage those risks. From my experiences, the ideal ratio has been around 1 onshore to 3-5 offshore.

Conclusion

Managing a global team has a lot of challenges, however, at the same time, with proper processes in place and good planning ahead of time, it can offer great rewards as well. I am hoping this paper offers you some ideas with which you can optimize your offshore teams. Along with globalization, accessing pools of highly skilled talent around the world has emerged as a new key strategic driver. As Lewin and Heijmen said, offshoring is "an intermediary step to new transformational global capabilities" (2008).

Reference

Henley J. (2006). Outsourcing the Provision of Software and IT-Enabled Services to India, *International Studies of Management and Organization*, 36, 111-131.

Lewin, Arie; & Heilmen, Antonius C.M. (2008). Offshoring: An Intermediate Step to New Transformational Global Capabilities: Findings from the 2007-08 Offshoring Research Survey, *Webcast* July 15, 2008.

Manning, Stephan; Massini, Silvia; & Lewin, Aroe (2008). A Dynamic Perspective on Next-Generation Offshoring: The global Sourcing of Science and Engineering Talent. *Academy of Management Perspectives*, Aug. 2008 (p35-54).

Lessons Learned About Distributed Software Team Collaboration

Raleigh Ledet
Apple Inc.
ledet@apple.com

Kal Toth
Portland State University
ktoth@cs.pdx.edu

Abstract

This experience report summarizes the conduct and outcomes of a practicum project [1] completed under the auspices of the Oregon Master of Software Engineering (OMSE) Program at Portland State University completed during the winter and spring of 2010.

Software engineers and learners are increasingly working in teams that are widely dispersed, often globally. Operating in distributed teams significantly increases the complexity of already complex software engineering tasks. Challenges amplified because of team distribution include resolving time-zone and cultural differences, addressing the need to become familiar with the tools and the human processes being used, and dealing with conflict within and across teams. This paper describes a practicum project conducted in a hybrid learning mode that studied one particular aspect - namely, the problems and properties of distributed collaboration processes used by software engineering teams.

The overall goal of this practicum project was to provide advice and guidance to distributed software teams, and to offer suggestions for further work in this field. The primary objective was to identify, specify and evaluate selected software engineering processes adapted to support collaborative software teams. The project was aimed explicitly at processes over tools - the software tools being the means rather than the ends for learning about such team collaboration. The OMSE students conducting this practicum project leveraged their industry experience and the software engineering principles and processes they had studied in the OMSE program. They progressively evolved a practical framework for defining and evaluating a specific core group of distributed software collaboration processes.

The ultimate outcome of this practicum project was that the students learned a number of critical lessons about how various distributed team processes and tools fit with the challenges of collaborating in teams. The students also confirmed that when working in a distributed team, be that as a practicing software engineer or part of a group learning exercise, the types of processes and tools that they use day-to-day are virtually the same. The practicum project described in this paper is an example of how software engineering practice informs learning as well as how software engineering education informs practice.

Biographies

Raleigh Ledet is a software engineer at Apple, Inc., operates a small shareware company called Mage Software, and holds a Master of Software Engineering degree from Portland State University. Previously he was with Wacom Technology Corp. of Vancouver, WA and DOGPAW, a non-profit organization.

Kal Toth is the Executive Director of the Oregon Master of Software Engineering Program (OMSE) and Associate Professor in the CS Department at PSU. He holds a Ph.D. from Carleton University in Canada and has worked for Hughes Aircraft, CGI Group, Intellitech, and Datalink Systems Corp.

1. Introduction

This experience report of the winter/spring 2010 practicum class [1] was performed under the auspices of the Oregon Master of Software Engineering (OMSE) Program at Portland State University. Offering traditional in-class software engineering courses since 1998, OMSE committed to offer all courses online in 2006. All OMSE courses are now available in either online or hybrid (face-to-face + online) delivery modes. However, some improvements are still needed.

Recognizing the common challenges of online learning and distributed team collaboration, several distributed team projects have been proposed to students for their (two-term) practicum projects over the last two years. OMSE practicum students have enthusiastically explored this area of "e-collaboration" [6], [7], [8] because they encounter such problems all the time on the job when working in dispersed teams. Some OMSE students have also published and presented papers related directly to the topic of distributed teams at PNSQC 2009 [4] and [5]. OMSE adjunct professor Milhauser in [3] explored several fundamental issues about distributed team collaboration some of which were addressed by this year's practicum teams. And Hines in [2] discusses the problem of conflict in distributed teams which informed this year's practicum team and therefore this experience report.

During this practicum the students explored distributed team processes through a combination of individual, online, and collaborative study and experimentation. They reviewed relevant literature, drew on what they learned about software engineering processes, and worked as a collaborative distributed team, the purpose being to reflect on and learn from their experiences. This enabled them to validate and criticize their various assumptions, draw reasonable conclusions, and make practical recommendations to others.

This year's practicum cohort consisted of eight students split into two cooperating sub-teams with slightly different distribution characteristics that adopted distinct, but related, project responsibilities. Similar team-of-teams of varying sizes are not uncommonly found in practice. This thereby offered a unique opportunity to realistically emulate day-to-day activities such as project management meetings, specification document reviews, and code inspections.

The emergent and evolutionary nature of the project informed the targeted results. More specifically, the students experimented with different collaboration processes and tools during the specification and planning phase of their work which fed the implementation phase. This focused their attention on some of the distributed team collaboration building blocks – both base processes and representative foundation tools. Some of these were rejected, while others were kept and incorporated into their emergent processes. In other words, the students "ate their own dog food" in an evolutionary fashion as they progressively evolved towards their final work products.

2. Establishing Teams

The project started with a clean slate without limitations as to which processes would be explored and defined, or which tools would be used. The joint team was tasked to develop a project specification and development plan to meet the practicum objectives, and utilize tools to shape and evaluate the fruit of their work.

The eight students in the class are all working software professionals with considerable hands-on experience developing software products and services. Two of the students were obliged to participate remotely as they lived and worked further away than commuting would allow. The furthest student lives in central California. Additionally, for the first three weeks, another team member participated from Italy while on a work assignment for their company.

The team members shared many traits including similar social and ethnic backgrounds, language and time-zone - other than the 3-week Italian connection. However, there were also several differences in corporate software engineering and organizational culture, work roles, responsibility levels, experience,

application domains, and, of course, personalities. Some of the students had worked together on class projects in the past. However, several had not.

One of the first things this practicum cohort accomplished was to organize into two teams of four persons each. Given that two students were obliged to attend remotely, it was decided to form a “local team” (Team A) and a “remote team” (Team B). One of the students would have had to make a long commute to campus and therefore elected to be on the remote team. Another member volunteered to work with the remote team. It was acknowledged that even the “local team” members would need to communicate among themselves electronically - they too were a distributed team when collaborating from their homes or workplaces. The team make-ups were set.

3. Deciding Project Goals

The practicum instructor proposed that the practicum students tackle a project related to distributed team collaboration. A software engineering assessment project was also on the table. Soon the students decided to consider the area of distributed teams. They were given considerable latitude in shaping the exact nature and scope of their project. After some brainstorming, the top topic choices were “assessing distributed collaboration tools” and “studying distributed collaboration processes”.

When deciding on projects, each OMSE practicum team normally decides on their own project and the various practicum teams work independently of each other. This year, with the instructor’s approval, the two teams decided to combine their efforts on the same over-arching project. There would still be two teams, but they would collaborate to divide the work and accomplish a combined project. As it turns out, this led to an unexpected distributed team collaboration challenge – namely some “conflict” - much like that which one would encounter on the job. Fortunately, the conflict experienced was very civilized and issues were resolved for the benefit of everyone involved.

4. Working with the Existing Distributed Infrastructure

OMSE courses are typically conducted in a hybrid delivery mode where face-to-face sessions are audio-video (a/v) recorded so that students taking the course online, or those who have missed a class for personal or business reasons, can stream the sessions to their desktops. Course content and class collaboration is supported by email and an online learning management system (Blackboard). The streaming video has approximately a ten-second delay. For regular lecture-style classes this is not a problem as online students do not view the stream live, but instead watch it at time convenient for them. Mind you, they do miss out on the opportunity to interact with the instructor and their class-mates.

A practicum class is not conducted like a lecture. The students need to regularly interact with each other in real time as well as with the instructor, but on a less frequent basis. This was the first practicum class that included remote participation and some modifications to the lecture tools were therefore in order. The first attempt used a telephone conferencing bridge with speakers and microphones in conjunction with the existing a/v streaming system. The stream was used to visually share content, namely, “talking heads” of the instructor and the students in the classroom as well as shared documents (like specs and plans). However, the video was of marginal quality and the delay was particularly frustrating. The telephone bridge also presented some challenges. The audio from the remote participants was routed into the room’s PA system. Although there were microphones in the classroom it was often difficult for both remote and local participants to understand what was said. Background noise would sometimes interfere with meaningful information exchange and drop-outs were sometimes experienced.

Over the subsequent few weeks, the team experimented with various setups and types of microphones, however, the improvements were not particularly satisfactory. Everyone was highly motivated to improve the collaboration environment and tools so that effort could be focused on the collaboration processes rather than the infrastructure.

5. First Collaboration Attempts

In an effort to give the newly formed teams an initial workout, and to come up with a tentative artifact as an anchor for discussion, both teams took a week to generate a straw-man requirements document.

Even though Team A was slated to be the local team, they soon decided to work in a distributed fashion and only meet face-to-face once a week right after class. They used Skype for team discussions and initially communicated by exchanging Word documents via email. This was a familiar and practical approach for collaborating as a team.

Team B experimented with Google Chat, Google Wave and Google Documents. While some members of the team were not familiar with instant messaging or Google Chat, none of the team members had ever used Google Wave or Google Documents before. This resulted in some pain while learning how to use these new tools and determining their limitations and trying to achieve some progress. Team B alternated between synchronous editing / discussion sessions and asynchronous editing sessions. Email was only used for notifications of changes for review or to setup a synchronous session.

When the two teams met again for the regularly scheduled Tuesday night class, they decided to use Team B's document as the basis for further discussion on the project goals. Each team shared what tools they used and how well they worked. Team A preferred synchronous voice meetings while Team B had a preference for instant messaging / text chats. Interestingly, Team B did not favor Google Wave or Google Documents. The main complaint with Google Wave was that the formatting capabilities left much to be desired. While Google Documents allows simultaneous document editing, the open sharing approach implemented by this tool allows someone to inadvertently throw away someone else's changes which proved to be problematic.

6. Project Challenges

As the project moved forward, the team refined the classroom setup, stopped using the streaming video capability and started using Google Documents as a shared Document medium. One person could share a Google Document and make edits while the document was displayed on the projector in the classroom or in the remote participant's browser. Unfortunately, the team members were at the mercy of Google Documents deciding when to synchronize all the viewers. Often this did not work and they would have to manually refresh the document. Google Wave did a much better job with live synchronization, but the lack of strong formatting capabilities hampered its use. The teams agreed that Google Documents was a great improvement over the delayed streaming video approach used initially. However, the team yearned for better tools while experimenting to determine the internal processes that work best for each team.

While struggling with the classroom tools, the teams were also struggling with the exact nature of project. They didn't have a clear, concise vision at this stage. After each class some form of consensus seemed to have been reached, yet attempts to work on the requirements document were mired in confusion, uncertainty, and disagreements on what had been decided in the previous class meeting. Getting eight mostly distributed individuals to arrive at the same understanding of the concepts being described is difficult at the best of times, and remained a challenge throughout much of the project.

7. Writing a Specification, Making a Plan

During the next class the team spent the entire three-hours working on project scoping and requirements. On this occasion, the team wrote a one-line description of the project's overriding goal which was:

To provide a set of processes to support software engineering throughout the SDLC [software development lifecycle] in a distributed collaborative environment and provide guidance for process selection for specific teams, project types and methodologies. [1]

The joint team also reviewed the current requirements specification and decided on guidance to be followed for each section and overall project scoping. Team A was assigned the requirements specification and Team B the Project Management Plan (PMP).

Being distributed 6 out of 7 days during the week continued to delay the formation of a shared context [1] for the class. The guidance that the team had spent three hours developing was found to be inadequate because it relied on a shared context that was not defined or available for consistent interpretation. At this point, the teams still had no leadership structure in place, that is, individual(s) responsible for clarifying, mediating, and arbitrating critical issues to facilitate and ensure that project momentum and progress would be maintained. The task conflicts similar to those anticipated in [1] occurred, evidenced by the "message churn" on the email distribution list. At this early juncture, the teams even disagreed over what constitutes consensus and the teams were still debating project goals.

The result of all this was that there was much uncertainty about what should go into the requirements document. Nevertheless, a first rough draft of a generic project management plan was developed recognizing that this could not be finalized without nailing down the requirements.

The practicum instructor strongly suggested that the teams rotate roles as the project progressed. The Project Management Plan (PMP) therefore defined four roles: Team Lead, Project Manager, Quality Assurance and Architect. Both teams implemented these four roles. The PMP defined one planning iteration and three work iterations and it was agreed that members of both teams would rotate into each role once.

During the subsequent class session, the class focused on developing clear definitions of the project goals and deliverables. The joint team agreed that the project should develop a set of processes designed to guide software practitioners of a typical product development firm when collaborating as a distributed team. The practicum teams would deliver a set of processes with usage guidance packaged in a form of an off-the-shelf product.

The following is a sampling of the processes that were addressed by the project:

- Synchronous Communication
- Synchronous Artifact Collaboration
- Asynchronous Communication
- Asynchronous Artifact Collaboration
- Artifact Review and Feedback

The teams also ratified the role-rotations and the iteration schedule including the assignment of team leads for the remainder of the planning iteration. It was agreed that the leads would work together to resolve project issues as they arose and the rest of the project management plan was reviewed and updated. With an established leadership structure in place and a plan of action to develop the requirements specification and refine the project management plan, the two teams were able to begin working independently on their assigned tasks and artifacts.

8. Collaboration Environment (tools)

Up to this juncture, the teams had not explicitly agreed on which tools to use to support the project. Team A had begun to use Google Documents for collaboration. Each team member was assigned a section of a document and team members worked asynchronously when refining a given document. Team A meetings continued to be held face-to-face after the joint team meetings, followed by sparse email and Skype communication during the week. Team B meanwhile continued to use Google Documents, text chat and email eschewing the use of Google Wave.

The practicum class had at its disposal an OMSE server supported by the CS department IT support group running a "Trac" wiki installation. Trac is a project management system that includes an open source project repository with lightweight project collaboration features including a generalized defect tracking system (a.k.a. "ticket system") and a source code control system (Subversion) integrated with Trac. It addressed two project needs: the ticketing system provided a central location to retain project tasks and artifacts including feedback about them; and Subversion provided a storage and revision control and tracking capability. These features satisfied many project requirements as well as the OMSE program's commitment to maintain project archives for future practicum students. It was well-understood that this asynchronous revision control capability would enable a much more reliable process than attempting to coordinate project artifacts by way of email. However, to continue using Google Documents would require exporting project artifacts to documents that the Trac wiki could store. It was therefore decided to drop Google Documents for a more familiar tool, namely, Microsoft Word.

However, the team still did not have an adequate tool to support distributed synchronous collaboration for the purpose of sharing and reviewing project documents. Fortunately, an alternative was introduced late in the winter term of the project. Portland State had acquired a cross-platform web conferencing tool called Elluminate that satisfied this critical project need. With simultaneous screen sharing, chat, and audio conferencing capability, Elluminate completely replaced the old streaming and telephone conferencing systems. The screen-sharing feature of Elluminate allowed the teams to easily share real-time updates of virtually any type of document being edited with all the formatting prowess of familiar native desktop applications. Though only the person sharing their PC screen could modify the document, this turned out to be much better for synchronous document collaboration than Google Documents or Google Wave.

All team members agreed that Elluminate was a marked improvement, however, certain issues remained to be resolved. Microphone echoing issues and displaying documents on the classroom PC and projector did not work seamlessly, marginally dulling the team's enthusiasm for Elluminate. To resolve these issues, the local students started bringing their own computers to class and joined the Elluminate conferences using their own headsets. Unfortunately, hearing the others in the classroom a second before hearing them again on their headsets proved to be distracting. Eventually, the most trouble-free solution was achieved by having everyone attend the class remotely from each other using Elluminate. Finally, the team discovered an approach whereby everyone in the class was able to collaborate effectively.

9. Introducing and Refining Collaboration Processes

As the Project Management Plan evolved, the teams decided to define processes and guidelines for conducting class meetings including creating agendas, keeping minutes and capturing action items. The minutes eventually made it into the Trac wiki, and the Trac ticketing feature was used to manage action items. Interestingly, the teams continued to use Google Documents and Google Wave despite some of their shortcomings.

Because an agenda is transitory and never meant to be formally captured, Google Documents was used to share agendas before meetings and allow team members to update them as required. Agendas needed some formatting features of Google Documents that were easier to use than those of Google

Wave. The minutes, on the other hand, did not need any formatting. However, to participate in their creation and updating them in real time, Google Wave fit the bill quite nicely. After the meeting, the scribe would deposit the minutes into the Trac wiki and create new tickets to capture action items. As the project progressed and mutual trust among members grew, the team discovered that they really didn't need to use Google Wave to capture and share the minutes - this task could be safely delegated to the assigned "scribe". Nevertheless, it became accepted practice (habit perhaps) to use Google Wave for this purpose. Once Elluminate became available, this tool's document sharing feature enabled sharing and annotating agendas and minutes as required during the meetings – which proved to be a very productive process.

By the end of the planning phase, Team A continued to use Google Documents for brainstorming and initial drafting and editing. They observed that assigning sections to various team members resulted in a document that lacked continuity and flow as the writing style changed from section to section. To help overcome this problem, they modified their workflow to allow each team member to work anywhere in the document. Instead of modifying someone else's work, they would add a comment in the document and allow the original author to make the modification (comments could be in both substance and style). Both teams encountered difficulties with Google Documents when trying to work simultaneously. They soon decided to avoid working simultaneously on their Google Documents.

Team B was somewhat frustrated with Google Documents. Google Document's formatting capabilities were sufficient but the team found it difficult to set up and maintain the consistent formatting they desired. Furthermore, concurrent editing resulted in lost work, and conversion to and from Word documents failed to meet expectations. Soon Team B switched to Microsoft Word retaining documents in Subversion for team sharing purposes. Since Subversion cannot auto-merge Word documents, Team B relied on the locking features of Subversion to ensure that only one person could edit a given document at a time. Generally, one person was assigned to write an initial straw-man document, and then the team would meet using Elluminate for a synchronous editing session. One team member was assigned the role of editor and made all the discussed changes to the document while the rest of the team observed via Elluminate's screen sharing feature, providing feedback when needed. Trac tickets were issued at the end of the meeting to assign unresolved fixes for later asynchronous editing.

Team B kept in constant communication during the week via email, instant messenger and Trac tickets. Team A held synchronous meetings using Elluminate to resolve outstanding issues with a given document assigning someone to convert the document for storage into Subversion.

By the end of the first practicum term (winter), the Project Management Plan was base-lined and both teams were beginning to follow its management processes. Along with the defined team structure, the teams were better able to re-solve their task assignments and the occasional conflict that would inevitably arise. Both intra-team and inter-team shared contexts were strengthening and the class became increasingly productive. The processes that the teams were organically discovering would become the basis for the processes that they would start executing during the next (spring) semester. In retrospect, the first term of the practicum was an essential and worthwhile process discovery and team-building exercise.

10. Iteration 1: Full Steam Ahead – A Few Bumps on the Road

After a short break between the winter and spring semesters, Teams A and B met up again to start working on the plan to develop the deliverables that were defined the previous semester in the specification document. There were still a few major decisions that needed to be made that the newly appointed team leads tackled the first week back after the break. The teams planned to both write and evaluate the collaboration processes to be written. The first draft of the document defining the process evaluation criteria had been created during the winter semester. This document needed some major rework, review, and ratification by the teams. Given the practicum team had eight process authors, they also needed a template document that would define their look and feel to ensure consistency across all of the processes.

In addition to the evaluation criteria and process template documents, each team constructed one process using Microsoft Word consistent with the process template. However, with over thirty processes in the requirements specification, some team members started to become concerned about maintaining consistency using Microsoft Word to write these processes. What if some aspect of the process template specification was changed? Would it be possible to consistently apply those changes to all existing processes? Emails were exchanged and a couple of solutions were suggested. Team A suggested the use of the Eclipse Process Framework (EPF) that was used on one of the OMSE classes. A member of Team B decided to build a web-based prototype tool that would take xml fragments and generate web pages that matched the look-and-feel of the defined process template document.

During the second class meeting of Iteration 1, the teams debated these two tools. EPF is powerful, free and generates attractive HTML output. However, there is a significant learning curve to achieve proficiency. And, even though EPF is based on the cross-platform Eclipse project, EPF itself only works on Windows and Linux. Two of the students had Macintosh computers and could not run EPF without a virtual machine (VM) installation which was considered cost-prohibitive, and some members have had sub-par experience with free VM tools. Meanwhile, the proposed custom tool was simple to use and easily customizable, but it was only a prototype at this stage. And there was concern that the team member proposing it would not have the time to create it or support it – after all, the project focus was to create processes, not develop a process-writing tool.

When it came time to decide on which tools to use, all members of one team voted for EPF and the other team opted for the prototype tool. The class was divided along party lines. The team did agree, however, to follow the leadership plan adopted earlier to resolve conflicts. The two team leaders met, discussed the issues, and finally resolved the deadlock with a coin toss. The custom tool was chosen.

The custom tool implementer was able to crank out the initial version of the tool with documentation within a few days allowing everyone to start writing their processes. By the end of the week the tool was generating a deliverable website, and the teams had written an additional eight processes and converted the two already existing ones using the custom tool format.

The following week was devoted to evaluating the processes that had been written. The plan was that each student would be assigned to lead the evaluation of a process written by the other team. Along with two to three other students, the evaluation team would role-play a scenario using the process which ensured that that each student would evaluate at least three processes. This turned out to be much more work than it took to create the processes, and there was a fair degree of push-back from the team.

11. Iteration 2: Quality over Quantity

The beginning of Iteration 2 gave the teams an opportunity to take stock of what they had accomplished and make necessary adjustments to the project plan. They flagged the following two major issues and discussed them at length.

The first issue was the question of quality versus quantity. Over thirty processes were defined in the requirements document and the teams had successfully written ten processes during the previous iteration. If the current pace was maintained, all of the allocated processes could be specified – at least in theory. However, it became clear that the quality of the written processes was seriously lacking. To write all the defined processes, the teams would not be able to spend much time on each process much less revise or fix any defects. Furthermore, it was agreed that the current pace could not be maintained. All agreed to modify the plan by limiting project scope to a small set of well-polished processes. The processes defined in the requirements document were divided into the following groups: “general”, “project management”, “requirements” and “development”. It was decided that the “general” processes would be the best place to focus the team’s efforts as these are referenced by processes in the other groupings. In addition, the ten general processes had already been drafted (though lacking) during Iteration 1. The new agreed-upon plan was to iterate over the processes already written to elevate them quality as much as possible.

The second issue was the evaluation criteria and process that had been defined. It was now recognized that even with a reduced number of processes to work on, role-playing the processes was not feasible. And the quality of the results that could be achieved from such an exercise was challenged by more than one team member. This issue was resolved in two ways. First, the formal role-play aspect was dropped and evaluations became an intellectual exercise performed by each assigned individual. However, the requirement to have the evaluation of a process performed by a member of the opposite team was kept. Second, the formal role-playing activity was replaced by an informal “eating your own dog food” strategy. This was possible because the general processes the teams were defining were exactly the same types of processes that were being used in the conduct of the project - for example, processes to synchronously or asynchronously conduct meetings and edit artifacts. The teams agreed to follow the fledgling processes during their collaborative project activities while updating them in parallel. This provided another means for team members to identify defects in their processes and record tickets for later resolution.

With this modified action plan the teams went to work polishing their processes. Each team member took one process per week and reviewed and revised it within their team before turning it over to the other team for evaluation. Both teams used the Trac ticketing system to assign work but otherwise continued to work as they had at the end of the winter term. Team A continued to use Google Documents to create draft processes which they converted to text files, placed in Subversion, and processed through the custom process specification tool. Meanwhile, synchronous meetings were held with Elluminate for desktop sharing; and Skype was also used because some team members felt that Skype provided better quality of audio than Elluminate. Extensive use of Elluminate, email, instant messaging, Subversion and the custom tool was made during this iteration by Team B.

12. Iteration 3: Finishing Up

The beginning of Iteration 3 provided a final opportunity to take stock of what had been accomplished and make necessary adjustments to the project plan. This time, the team identified four critical issues that needed to be addressed.

Applying the custom tool and process template document enabled all of the processes to exhibit a common look. However, at the end of Iteration 2 there was one lingering consistency issue. Each process was characterized by a workflow of re-usable work tasks. A work task may or may not take some artifact as an input; and after performing the steps of the work task some output artifact is created. In essence, a work task is a foundation process element that cannot be broken down into smaller chunks, and a process is a unique ordering of work tasks with some additional supporting text. Team A and Team B differed on when an activity should be expanded into a complete work task, and when it could be written as supporting text of the process. The team leads considered the issue and decided that this inconsistency in the processes was not great enough to warrant changing at this point in the project.

While the custom tool happily created a web site composed of a collection of process pages, it was missing standard document trappings such as an introduction and a usage guide that would bind the pages into a cohesive deliverable. A minor unofficial effort to resolve this issue was undertaken during Iteration 2 and two team members were officially tasked to finish this in Iteration 3.

Along with delivering a collection of processes, the practicum team planned to deliver a document that mapped which processes were best-suited for the various stages of the software development life cycle. It was decided to deliver this document even if this implied devoting less time to polish existing processes.

The teams also needed to allocate effort and schedule to create a number of deliverables during this final iteration to meet the requirements of the practicum project including a final presentation and a final report which incorporated the project’s specification, project management plan, process evaluation criteria, and process deliverables. This last iteration required a lot of cross-team collaboration to generate these project deliverables and prepare for the final presentation.

The class decided to use the Elluminate tool, together with other tools they had used, to conduct a distributed team presentation of their project accomplishments. Conducting a dry-run a week before the final presentation proved to be invaluable. The team used the experience it had gained with the collaboration tools and distributed collaboration processes to setup the classroom such that they would avoid many technical and process problems during the final presentation.

The final presentation was performed using Elluminate's document-sharing and white board features to display PowerPoint slides summarizing the project. A PSU classroom on campus was secured and potentially interested parties were invited to attend. This included current and past OMSE students, faculty, and industry guests. Team A was present in the classroom together with the instructor and some of the invited guests. The Elluminate session was projected on a large screen and the audio was piped into the classroom PA system. The local team also had their laptops linked into the Elluminate session through the WiFi network. Team B and several other invited participants attended remotely via their PCs and headsets. The order of speaking and audio had to be carefully controlled by the team, each member having a critical role to play in both controlling and making the joint presentation.

13. Summary

This practicum was a hybrid online learning experience conducted in a university setting. Interestingly enough, the challenges faced by the students were virtually the same as those encountered by distributed teams in the real world. In both educational and industry settings one would ask questions like: "How does a distributed team organize a meeting?" "How do team members work on and review project artifacts like specs and plans?" "How does one team interact and work with another team?" "How do team members communicate facts and issues to other members?" And, "How do distributed teams ensure that everyone on the team has the same interpretation of project information?"

In general, the practicum teams found that they could communicate effectively in an informal fashion when the team was small (e.g. four). However, joint meetings of all eight participants required a more organized and systematic approach. In addition, getting eight team members to adopt and follow the same vision for the project did not turn out to be a simple matter. The lack of an empowered subset of the team with a clear understanding of the vision and good communication skills undermined early achievement of goals for the team. Once a leadership structure was put in place decisions were faster and progress was more tangible and visible to the entire team. The planned iterations facilitated positive forward progress because they allocated specific effort and milestones to achieve clearly articulated iteration goals. The teams were able to continue to make progress following an imperfect plan knowing that unresolved issues would be addressed at the beginning of the next iteration.

However, certain aspects of the team's process made it more challenging to keep everyone motivated. The lack of face-time and non-verbal communication cues made it difficult to tell when someone was no longer engaged. Furthermore, it was found that when such cues were missing from the communication medium, messages were easily misinterpreted potentially leading to misunderstandings and even ill-feelings amongst team members. And such disconnects tended to go un-noticed for longer periods across distributed teams than collocated teams. Addressing them as soon as possible is always a good idea.

Team building and team cohesion strategies where not explicitly investigated. However, it soon became clear that distributed teams need to seek out ways to build trust and camaraderie to overcome interpersonal communication problems. The practicum teams overcame some of these issues to a degree by encouraging each team to explore how they best worked well together.

Finally, when it comes to distributed collaboration tools, to no one's surprise, it was verified that no one tool fits all requirements. A distributed team needs to use multiple tools to work together effectively. Furthermore, the teams discovered that it is highly beneficial to have certain team members become proficient in their tools of choice acting as technical support and consultants to the other team members.

Both teams experienced some pain points early on which were overcome as the project progressed. After reviewing Hinds [2] p. 617-623 later in the practicum, the teams reflected on their experiences and thereby came to a better understanding of their "issues". As explained by Hinds on the topic conflict on distributed teams, a new team needs to develop a shared context for working together. Early in this practicum, some of the team members encountered a degree of internal conflict about the central focus of the project and their particular roles in this project. Their initial results were less than hoped-for with fairly sparse coverage of the topics – and they wondered why. But this could be explained by the initial lack of shared context and lack processes in place. Once the teams began to understand their true goals and develop processes for moving forward, they jelled as a team, fell into a rhythm, and began to achieve step-wise successes. Although they constructed an initially over-reaching specification and plan, they were able to hunker down, de-scope the project, and achieve a stellar result.

14. Conclusion

The OMSE students conducting this practicum project were well-aware of many of the challenges of working in distributed teams and were able to leverage their industry experience in this interesting problem area. They were able to combine this experience with the software engineering principles and processes they had studied in the OMSE program. And, by way of thoughtful and systematic experimentation, they were able to progressively evolve a practical framework for defining and evaluating a specific core group of distributed software collaboration processes.

The ultimate outcome of this practicum project was that the students learned a number of critical lessons about how various distributed team processes and tools fit with the challenges of collaborating in teams. The students also confirmed that when working in a distributed team, be that as a practicing software engineer or part of a group learning exercise, the types of processes and tools that they use day-to-day are virtually the same. The practicum project described in this paper is an example of how software engineering practice informs learning as well as how software engineering education informs practice.

Acknowledgement

The referenced OMSE software engineering practicum team was comprised of Thomas Carlier, Jason Cowley, Adam Kolb, Merri LeClerc, Raleigh Ledet, John McConnell, Grant McCord, and James Thompson. OMSE faculty member Kal Toth facilitated the project. The authors of this paper thank the team members for the valuable outcomes that emerged from their work as well as their review and thoughtful contributions to the creation of this paper.

References

[1] Carlier, T., J. Cowley, A. Kolb, M. LeClerc, R. Ledet, J. McConnell, G. McCord, J. Thompson, "Distributed Software Engineering Process", Final Report, OMSE Practicum, winter and spring 2010.
[2] Hinds, Pamela J, and Diane Balley. "Out of Sight, Out of Sync: Understanding Conflict in Distributed Teams." Organization Science 14 (2003): 615-632
[3] Mihauser, Kathy, "Distributed Team Collaboration", PNSQC, October 26-28, 2009
[4] Eby, S., B. Rydell, C. Seaton, "Distributed Requirements Collaboration Process", PNSQC, Oct. 2009.
[5] Khan, A., R. Ramadass, L. Rosenbaum, B. Varma, K. Toth, "Distributed Collaborative XP Meetings", Poster Paper, PNSQC 2009.
[6] Khan, A., R. Ramadass, L. Rosenbaum, B. Varma, "Distributed Collaborative XP Meetings", Final Report, OMSE SE Practicum, winter and spring 2009.
[7] Eby, S., B. Rydell, C. Seaton, "Distributed Requirements Collaboration Process", Final Report, OMSE SE Practicum, winter and spring 2009.
[8] Bates, K., J. Maurer, S. Wainstock, E. Wilson, "Social Networking", Final Report, OMSE SE Practicum, winter and spring 2009.

Scaling Agile Practices; Replicating the small team achievements to larger projects

Don Hanson
Don_hanson@mcafee.com

Abstract

Have you ever played the telephone game where a sentence is whispered to the next person in line? It's amazing how much the sentence changes as more people are added to the line.

This game illustrates how working with larger groups of people introduces new challenges. Communication issues in this instance. Likewise larger software development projects can introduce new challenges for agile teams to overcome.

In this paper we'll look at different techniques for addressing three challenges commonly associated with larger projects:

- **Scaling the development team size** - It's easy to know what everyone is doing without taking up most of your day when there are three or four of you. This is much harder to do without spending more time as the development team grows to 10, 20 or more.
- **Big projects have big features** - You're much more likely to run into features which will require multiple iterations to complete on larger projects. How do you handle these iteration spanning features in an agile manner?
- **Reducing the post development end game** - External factors can often increase the time between completion of feature development and the software available for sale.

These approaches may help address similar issues on your projects.

Biography

Don Hanson founded his first company and began writing commercial software in the early 90's developing an evolving line of animation plug-in products. He has since worked on projects ranging from the mobile client for a wireless navigation startup to enterprise-level commercial software products. Don is the development manager for McAfee's flagship enterprise security management product.

Scaling Agile Practices; replicating the small team achievements to larger projects

Have you ever played the telephone game where a sentence is whispered to the next person in line? It's amazing how much the sentence changes as more people are added to the line.

This game illustrates how working with larger groups of people introduces new challenges. Communication issues in this instance. Likewise larger software development projects can introduce new challenges for agile teams to overcome.

In this paper we'll look at different techniques for addressing three challenges commonly associated with larger projects:

- **Scaling the development team size** - It's easy to know what everyone is doing without taking up most of your day when there are three or four of you. This is much harder to do without spending more time as the development team grows to 10, 20 or more.
- **Big projects have big features** - You're almost guaranteed to run into features which will require multiple iterations to complete on larger projects. How do you handle these iteration spanning features in an agile manner?
- **Reducing the post development end game** - External factors can often increase the time between completion of feature development and the software available for sale.

These approaches may help address similar issues on your projects.

1. SCALING THE DEVELOPMENT TEAM SIZE

It's easy to know what everyone is doing when there are three or four of you. It gets much harder as the dev team grows to 6, 7, 10 or more. An approach we've had success with is to focus on scaling team organization and communication. How we address the former influences our options for the latter.

Communication Overload
The style of communication and level of control possible on a 3~4 person team is dramatically different from what is possible with 10 or more people. Consider the different experience you have going to a restaurant with three friends compared to a party of 10 people.

Ever notice how larger parties start off pretty quiet, not much discussion going on? Then someone starts up a conversation and most people listen in, information distribution. Eventually, if you're lucky, it breaks up into two or three smaller conversations.

Conversations
The key to effective high-bandwidth communication on larger teams is to scale the number of conversations across the team while maintaining the number of people in any given conversation and the number of conversations each team member must participate in to do their job. This subtle but important distinction is a common pitfall for many teams.

If not recognized the early stages of team growth can result in a conflicting and very negative experience for team members. On a small team you're often more successful dealing with the statement of "There's no time to do my work" as a sign of resistance to high bandwidth communication. As the team grows we may hear the same statement for very different reasons. Our previous success can blind us to the reality that we're facing a very different situation.

We want to keep small groups of people focused on the same thing. We want that high band-width communication going on. We want people focusing and talking. For example, with 10 developers we may want 3~4 conversations going on the same time.

Feature teams
We know what we want but how do we get there? How do we empower small groups to have those conversations? We want each team member to spend most of their discussion time each week on features they are working on. If they are working on a feature then it should imply they have some level of control over it. The concept of feature teams are a natural fit.

A feature team is typically a cross functional group tasked with delivering shippable features and workflows. This differs from the traditional pattern where teams are organized by component or layer, e.g. one person handles the UI, another focuses on the database and so on.

Organizing the larger team into small groups of people with all the skills needed to deliver their feature combined with keeping the focus on delivering working features helps ensure each conversation has the right set of people involved and each individual is not overloaded with conversations that don't necessarily pertain to them.

Good Organization Is Critical
I come from a military family. The Table of Organization and Equipment (TOE), aka "org chart", was a concept we were familiar with at an early age. There's more to creating an effective organization of a large team than simply making it look like a pyramid.

Questions specific to your situation and product must be considered, such as how much oversight is needed? Are you making medical equipment or video games? Does your team work relatively alone or in tight coordination with others to deliver your product?

Currently our product engineering team consists of 10 distinct development teams that work together to deliver a single release; co-located server foundation, server application and client teams, and remote client team, each with Dev and QA teams, and design and tech pub teams. Some of these teams have doubled or tripled in size in recent years.

We've integrated a simple chain of command into our development process to enable the appropriate amount of high bandwidth communication and rapid decision making within the large server application Dev & QA teams, and between the other teams. There are three levels involved in the daily creation of software; feature teams, leads, managers & architects.

- Feature teams are comprised of one or more developers and QA, a Tech Pubs contact and a Design contact. Each feature team has a Dev and QA lead to represent the status of the feature area to others and be the point of contact for questions.
- Next up are the Dev/QA team and technical leads. This is a thin layer that handles immediate coordination, such as balancing user stories among team members within an iteration, and technical oversight of key areas, such as database performance. These guys are your master sergeants who keep the wheels turning.
- Finally we have the managers and architects, who share many of the same goals, have over-lapping interests and distinct but complimentary areas of authority. They tell the team what hill to take and the approach the team will use.

Not exactly rocket science so far. What has made it work well for us has been the focus on enabling decision making throughout the chain with appropriate review. We want higher levels to set broader directions while enabling more detailed design and implementation at lower levels. We want to avoid embarrassing I-wish-someone-had-told-me-it-worked-that-way mistakes while minimizing the time spent in reviews and meetings. And finally we want the team to be able to move forward while people are on vacation.

Some things we change during an iteration retrospective, for others we may wait for the next project. Every situation is different. For example, here are the current owners for some of the Dev team artifacts.

- Themes - owned by managers (who have more schedule visibility across the ten teams)
- Designs - owned by feature teams
- Stories & tasks - owned by implementers

Keeping In Sync
During the re-plan phase before an iteration we invest in two team meetings to help communicate project status in a meaningful way.

During the iteration demo the feature teams show off the new working features and workflows on running code on a demo server located in our intranet. The meeting is webcast and by request saved for 30 days for later viewing.

After watching a small but steady stream of wow-I-wish-we'd-thought-of-that-earlier suggestions raised by the wider project team during the iteration demos, we added an iteration preview working lunch. We order in some pizza for lunch and feature teams describe verbally or using storyboard/prototype what they will be creating next iteration

2. BIG PROJECTS HAVE BIG FEATURES

How to build those BIG features in an agile manner? A common pitfall for all Agile projects is losing the focus on iterative development and taking on an incremental approach. This can be hard to avoid on smaller projects but seems to go hand in hand on larger efforts.

With so much to do on a large project it's easy to fall into the trap of focusing on delivering "complete" chunks of functionality. Growing up we've learned to judge our progress on a particular task by how much we've completed. As we attempt larger and more complex projects outside our comfort zone it's hard to resist falling back on old habits.

It is worth noting that the path to incremental development is a slippery slope. At the project level iterative and incremental are not mutually exclusive. It is quite possible to use an iterative approach delivering one major feature and an incremental approach on another.

Incremental VS Iterative
The problem with delivering shippable chunks of functionality is that it requires the end goal to be well defined from the start of the project. One of the strengths of Agile development is that it allows the details of the end solution to be defined over the course of the project.

For example, suppose a project starts with the intent to build online community software with message forums but midway through the project it is discovered that more of a focus on a question and answer system is needed.

At this point with an incremental approach very few of the workflows are working so the impact to the overall user experience is unclear. In addition, a good deal of effort will have been wasted on "completed" components that now need to be changed or discarded.

However with an iterative approach, where most of the workflows are functional (if unrefined), understanding the user experience impact should be quite possible. The level of rework should be fairly minimal as impacted features are still being refined.

Jeff Patton from Thoughtworks has an excellent presentation on the differences between incremental and iterative development, along with strategies for dealing with external stakeholder's fear of uncertainty on what will ultimately be delivered. It is well worth the read and I highly recommend checking it out.

The focus here is on helping the development team maintain an iterative approach. Jeff provides an easy check to see if your project is iterative or incremental; "it's not iterative if you do it only once"[1].

Try a New Perspective
For a variety of reasons it can sometimes be challenging for an agile team to reorganize the incremental delivery of a feature along iterative lines. There are usually very reasonable sounding rationalizations as to why this particular feature is "different".

We've found that looking at it from the viewpoint of other stakeholder can help. A fresh perspective can make all the difference. Now where are we going to find other stakeholders that have the time, inclination and familiarity to offer advice? Feature teams to the rescue!

Fortunately delivering functionality iteratively does get easier with experience.

3. REDUCING THE POST DEVELOPMENT END GAME

Do months go by after feature development stops, aka feature complete, before customers can purchase your software?

Upper Management Buy In
The most important part of this process is getting buy in from upper management that changing the situation is a priority. It's amazing what can be achieved when everyone shares a common goal. This critical piece of support changes the problem from "how are we going to do this?" to "what's stopping you from doing this?"

Break Large Problem into Smaller Ones
To address this seemingly impossible problem we used the simple technique of breaking larger problems into smaller ones. Instead of trying to solve everything at once we separated out approach into three stages;

1. Define the specific problems faced by each stakeholder
2. Define a set of measurable, high level goals that would address those problems to the stakeholder's satisfaction
3. Create a detailed, measureable plan to achieve those goals

Big Systems Have Many Customers
Each stakeholder was first asked to identify the challenges preventing them from committing to reducing the end game by half, from six months down to three. In our case, a long compatibility testing cycle with 50+ products that integrate into our enterprise security management console was cited as one of the main reasons for our long end game.

Bring Quality Forward
Reducing the compatibility testing was considered too risky. Starting the testing earlier seemed the only viable option. To make this happen we needed to bring quality forward so the compatibility testing could begin prior to reaching Feature Complete (FC). This was selected as a key high level goal for Dev, QA, Design and Tech Pubs.

It's All in the Details
In a series of meeting the details of what the high level goal of reaching FC with higher quality means to each team was defined. Plans for achieving and measuring our progress towards these detailed goals were worked out stakeholder.

[1] Embrace Uncertainty: Strategies for on-time delivery in an uncertain world. 2008. <http://www.agileproductdesign.com/presentations/index.html>

In addition to the many specific measurable goals, certain process tweaks were included, such as forming the feature teams earlier to increase the calendar time available for the conceptual and high level design work. For Tech Pubs this means writing the conceptual descriptions for features much earlier in the project and detailing the workflows during a second pass later in the project.

With that we were on our way. Is this large process change worth doing? Absolutely! Is there risk associated with the change? Yes. (Or we would have made the change a long time ago) Do we have fallback plans for this large project risk like a good development project leadership team should? You bet we do!

From Day One

We were pleasantly surprised at the number of the end game processes are under our control. We went through the exercise of listing out all of the high level end game tasks and asking if they could be done "from day one" of the next project.

For example, upgrade and migration issues caused a number of headaches in the last release. However, migration is under the team's control. For the next release we will have automated upgrade testing working from day one! A number of you already spotted this and you're right; "day one" might mean week one or first iteration code drop. But the concept holds true.

If you think about it working upgrade from day one is probably one of the easiest stories in all of the first iteration. What does upgrade consist of when nothing has had time to change? It could be as little as the install wizard runs without graphics and the version gets updated.

With this approach we've turned a painful integration task into an early warning radar from day one.

Acceptance Tests Driving Development in Scrum

Sari Kroizer – Validation Team Leader

Sari.Kroizer@intel.com

Abstract

This paper demonstrates the challenges faced by the validation team when implementing Scrum, and the benefits gained by adding an Acceptance Tests Driven Development (ATDD) methodology. The paper will share with you the experience of our Scrum team on a large software project (23 sprints). The paper includes actual data on test coverage and lessons learned.

Author Biography

Sari Kroizer, Validation Team Leader and Scrum Master in Intel Israel, works with the team and the managers leading process improvements using Scrum and validation methodologies.

She has formal education in software engineering and has experience in programming, web development, and firmware/software validation.

1. Introduction

Validation testing as an integral part of the Scrum process is a big challenge. The cycles are short, regression testing is more necessary than ever, and quality is a requirement for story completion. Under these conditions, the testers need maximum flexibility and good tools.

The ATDD methodology provides a quick solution for integrating validation testing in Scrum. Automated acceptance test cases are designed and developed in parallel with or even before the features.

From our experience, using ATDD with Scrum significantly improves both the development and testing processes. The ATDD methodology helps the whole team understand and focus on the customer's requirements. This is a major goal of Scrum. ATDD helped us to combine the validation into the Sprints, to improve the quality, and to raise the team satisfaction. ATDD also improves the use of testing resource time, including optimization of tester personnel ramp up. And last but not least – it's really fun!

This paper includes analysis (with actual data) of the key problems we faced when implementing Scrum and how we solved them by adding ATDD.

I suggest implementing ATDD on long projects, where the automation is most valuable, and to run manual tests in addition to the automated acceptance tests.

2. QA Environment in Scrum

As the validation team leader, it was my challenge to adapt our validation process to the complex Scrum environment. This required a new mind set: No more long test plans, fully-coded features, or project-long tests cycles. In the Scrum model, the validation process must be as agile as the development process.

We decided to include the software testers in the Scrum team, and to perform validation as an integral part of the sprint. We also defined that the quality of a story is a prerequisite for story completion. This means that a full test cycle is done on each story as part of the sprint, and all the critical bugs and most non-critical bugs are fixed. The number of bugs a story can have at the end of the sprint is set according to the story size.

We also needed to take responsibility for the complete project validation, and to make time for integration and regression tests. In Scrum, the project is incremental, and the deliverable for each sprint is stable and validated software **with** the additional new stories of the sprint.

From the beginning, we recognized that we would need an automated regression test suite for our project. We hoped that we would be able to use this automation for future projects as well. We started to work on the new stories, and tried to build a set of automated tests to help us with the repeated regression cycles, in parallel to the full cycles. The Scrum team included five developers and five testers, and the tests cycles were part of the sprint. The duration of each sprint was three weeks.

Progress of the stories is recorded using a “board” (see Figure 1). In the two first days, the high level design documents and the test cases for the new stories are prepared and reviewed. When the developers finish implementing the new story, the story is moved to “Under Test”. The test cycle starts and has to be completed before the story can be moved to “In Debug”. After most bugs are fixed, no critical bugs exist, and the tester agrees, the story is moved to “Done”. We also had stories for regression tests. Bugs found in the regression test cycles, were scheduled to be fixed in the sprint. A quality criterion was defined for each sprint release.

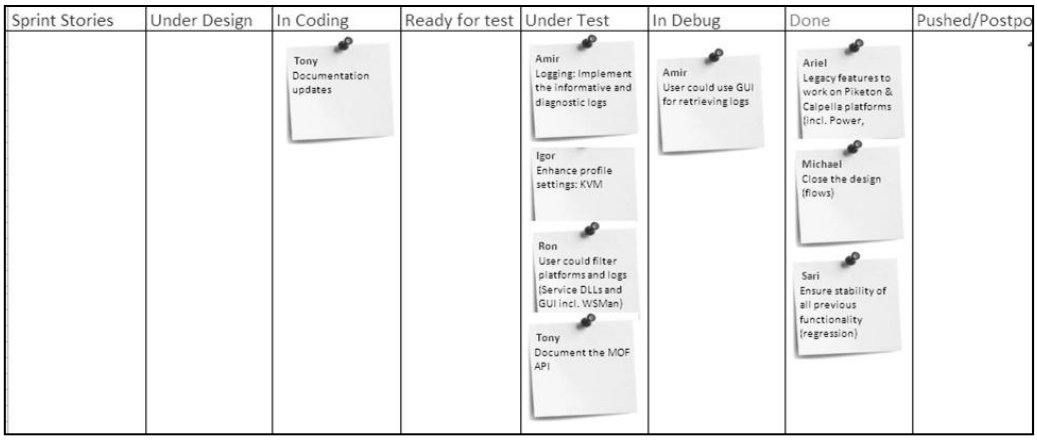


Figure 1: Sprint 13 Board – At the end of the Sprint

3. QA challenges in Scrum

This section describes the main challenges we found when implementing Scrum without ATDD.

3.1 Idle Time

During the sprints, team members often cannot start their own sprint tasks until other team members complete their tasks. This causes 2 main periods of idle time for team members:

- **Testers** – After the first 2 days of design and writing test cases, the testers must wait for developers to deliver new features.
- **Developers** – After delivering features to the testers, the developers must wait until the testers complete the first test cycles.

3.2 Wasted Time on Unstable Features

It can take a long time to stabilize new features. The developers complete code and deliver a feature and the testers start the test cycle. After some manual tests, the testers often realize that a main flow of the feature is not behaving as expected and is blocking other tests. The developers fix the issue and the testers repeat the test. If the feature still does not work correctly it must be returned to the developers again and the “ping pong” game continues.

Sometimes the code of a feature is completely changed during these iterations. If the testers try to continue with tests that are not blocked, they will need to repeat them again because the code has changed. In addition to wasting time , which is valuable since it's constrained to the sprint cycle, the process can be frustrating for the testers.

This situation occurs because developers, unlike testers, do not usually focus on how the customers expect features to work. Also, the testers sometimes do not focus on the high priority tests first.

3.3 "Mini-Waterfall"

One of the main problems is that the Scrum process can unintentionally become almost the same as this description by Hector J. Correa in "Introduction to Scrum":

"A common misunderstanding is to see the Scrum process as a mini-waterfall. Just having short iterations is not enough to make Scrum work. For Scrum to shine it is vital that all team members work together on the same goal during the Sprint. If developers are coding a set of features but QA is testing a different set of features they are not working together. You are not going to have potentially shippable code at the end of this Sprint"

If the team does complete a story in a sprint, the test cycle cannot complete on time. This can leave several stories in the "Under Test" status at the end of the sprint (see Figure 1).
Thus validation completion is postponed to the beginning of the next sprint and the sprint ends with code that cannot be released.

While the cycles are running, the developers are already focused on coding the new stories. Bugs from stories in previous sprints get second priority and the technical debt of unresolved bugs and uncompleted stories starts to increase.

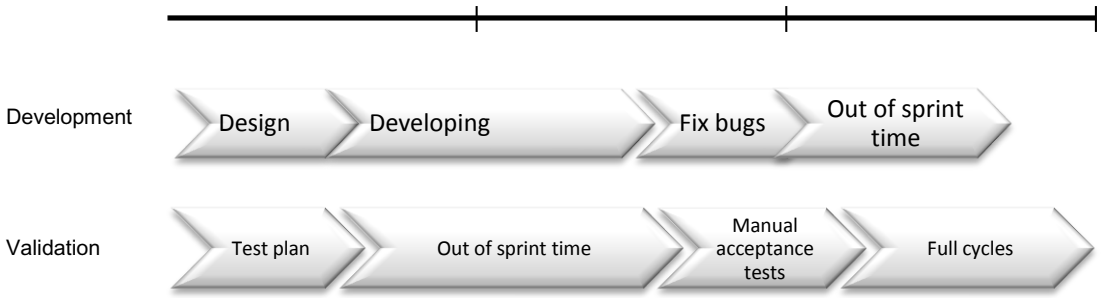


Figure 2: Gaps and Overflows in a Three Week Sprint

3.4 Missing Automation

The first priority and focus of a sprint is validation of new stories. Automation tasks are assigned to "free time", and the automation progress follow up is not part of the sprint tasks follow up. The validation team tries to make progress coding the automated tests in parallel to the sprint tasks. This mode is problematic.

To solve this problem we tried to add “Automation Enhancement” stories to the sprint backlog. Because they were not “real” customer stories they received low priority and were usually the first to be pushed out of the sprint. This caused slow progress in automating the test cycles and therefore we had to run the regression manually.

4. Adding the ATDD Methodology to Scrum

This section describes ATDD and how we implemented it in our Scrum process.

4.1 What is ATDD?

The Acceptance Tests Driven Development (ATDD) methodology extends the Test Driven Development concept to a higher level. In ATDD you start implementation only after you have defined the specific customer valued functionality you want the system to exhibit.

ATDD includes the following:

- **Acceptance Tests** – Specifications for the desired behavior and functionality of a system. For each story, an acceptance test defines how the system must handle certain conditions and inputs and the expected outcome. The tests are for expected results (behavior), not internal structure (methods).
- **ATDD Process** – Each story must include a clear acceptance criteria defined by the customer. The testers and developers define together the acceptance tests cases for the story. The next step is to automate the acceptance tests. These tests will fail until the story is implemented. When the story is ready, the acceptance tests must run and pass. The functionality cannot be considered as “implemented” until the acceptance tests pass.

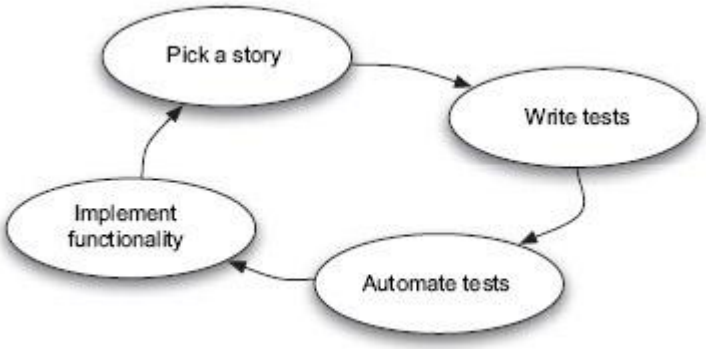


Figure 3: The ATDD Cycle

4.2 Our Implementation of ATDD

For internal reasons, the original Test Driven Development (TDD) methodology was not adopted by the developers. The validation team pushed for the adoption of ATDD in our process. We included the acceptance tests automation tasks as part of the validation tasks, and executed them in parallel with features development.

Preparation

The first effort was to set acceptance criteria for all the stories in the product back log. This task was done by the product owner, with help from the architect and the validation team leader. The product owner worked with the customers' representatives to validate the acceptance criteria, and to make sure that they fit the customers' needs. The completed acceptance criteria for the stories then served as the basis for our test plans.

The Sprint Process

In the first two days of the sprint the high level design of the stories and the test cases are written. During this time the acceptance test cases are designed as well, according to the acceptance criteria. These test cases are reviewed by the developers, and the entire Scrum team must commit to them. While the developers start to implement the stories, the testers start to automate the acceptance tests cases. The interfaces are implemented first. This allows the automated tests to use the interfaces even before the functionality exists.

We added a new column to our board: "code completed". When a story is fully developed we move it to "code completed". The automated tests are now ready to run and the testers start running them. Then the refactoring process begins. The full validation cycle is not started, until the story passes the automated acceptance tests. Then we move the story to the "ready for test" column. When the new features pass the acceptance tests the validation cycle continues with a full manual tests cycle, including less important flows, negative tests, documentation tests, etc.

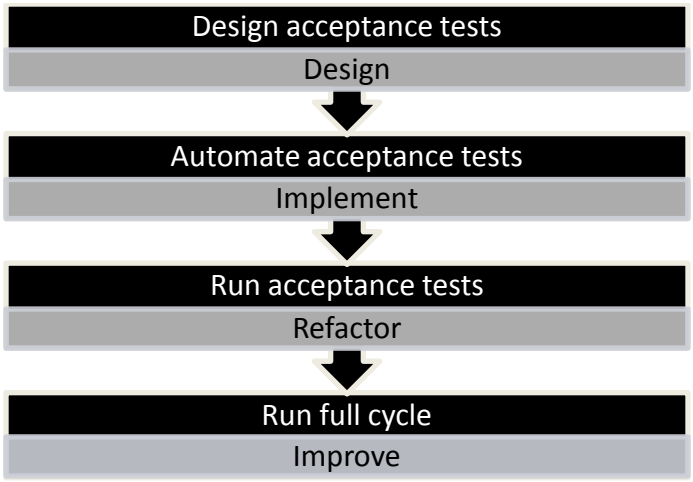


Figure 4: The Sprint Process

4.3 Example

This example demonstrates how we implemented ATDD in a story called "User could configure remote access settings to the firmware".

The acceptance criteria for this story were defined by the product owner:

1. *The user can set remote access settings including: address, port, trigger, certificate, and a strong password.*

2. *The user can configure and reconfigure the firmware using the remote access settings.*

The sprint started and this story was picked up. During planning, the high level design and the acceptance tests cases were defined for the feature. This table shows the acceptance test cases, according to the acceptance criteria (committed by the team):

Test	Description
1	Configure the firmware with remote access settings: a. Set the remote access settings in the service. b. Perform the service “configuration” command with the created settings. c. Check the settings in the firmware.
2	Change the remote access settings in the firmware: a. Change the remote access settings in the service. b. Perform the service “re-configuration” command. c. Check the settings in the firmware.
3	Delete the remote access settings from the firmware: a. Disable the remote access settings in the service. b. Perform the service “re-configuration” command. c. Check that the settings were deleted from the firmware.

We had more tests in the test plan: Advance functionality tests, negative tests on the settings fields, usability tests, system tests, checking how the firmware implements the settings, and documentation tests. (Note that the additional tests are not acceptance tests.)

While the developers were developing the feature, the testers implemented the automated acceptance tests. After one week the code was completed and the automated tests started to run. During the first test run, Test 1 passed but Test 2 failed. It took two days for the bug to be fixed, and some implementation changes were required. During this time, the testers continued to work on other stories, and didn’t waste their time on the full cycle of this uncompleted story. The automated test was run again and again, until the story passed.

After the acceptance tests passed, the testers started the full cycle and completed it in 3 days. Some bugs were found during the cycle, but no additional huge implementation\design changes were needed. The bugs were fixed before the sprint ended and at the end of the sprint the story was “Done”.

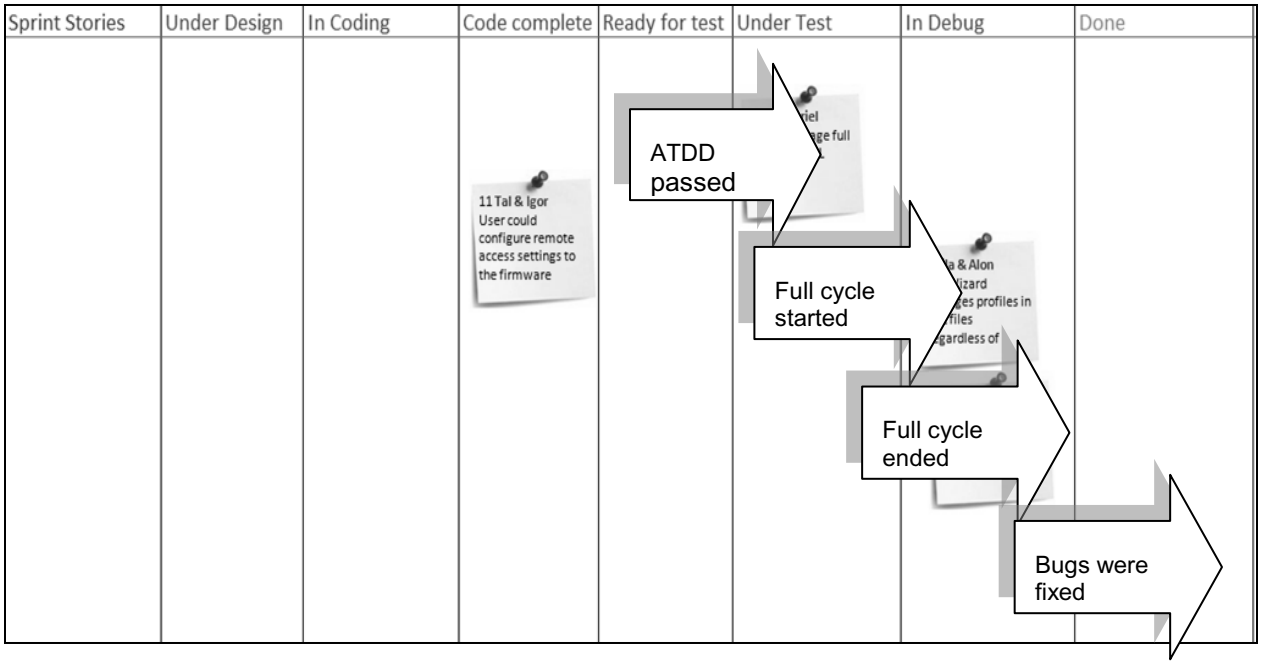


Figure 5: The Sprint Board with ATDD

5. Results

This section describes how challenges encountered in implementing the Scrum framework are solved by adding the ATDD methodology.

5.1 Less Idle Time

The validation team members use the first part of the sprint to automate the acceptance tests. While developing the automated tests they gain experience of the feature. This experience and the fact that the stories must first pass the acceptance tests also greatly reduce the duration of the manual tests cycle. While the testers do the manual tests, the developers fix the bugs and improve the code.

For example: In a sprint with 7 stories, before implementing ATDD the testers would have wasted time (at least 4 days) waiting for the features. With the ATDD, this time was exploited learning the new features and writing the automated acceptance test cases for each of the stories. 16 test cases were implemented and could be run automatically.

5.2 Less Wasted Time

Because the acceptance tests are focused on the customer's requirements and include the most important main flows, they give quick answers about the status of a story.

The testers that run the tests-already know the interfaces and are aware of the acceptance criteria. This means they don't waste time learning the feature during the tests. In addition, the acceptance tests are automatic, and are easy to re-run (and to improve them while running).

The focus of the developers on the customer's requirements is also improved. They know that if the acceptance tests fail, the story cannot progress to the ready for test status. Their code must address the acceptance criteria which are based on the customer's requirements. Also they generally run better sanity tests to make sure that the story will pass the automatic tests.

For example: Using ATDD, a story was tested in 6 days (2 days running ATDD and 4 days for the rest of the cycle). Without ATDD, this story would have taken at least 10 days to run.

5.3 Back to Scrum

Including ATDD prevents the team from slipping into a "Mini-Waterfall." All the team members work together on the same goal during the Sprint. The developers code a set of features and the testers test the same set of features.

The duration of the acceptance tests cycle decreases because the features are more stable and fit the requirements. The full cycle time also decreases because the testers have the knowledge and the experience, and the main flows already work. Most of the critical bugs are found when running the acceptance tests, and are fixed immediately. The rest of the bugs are found and fixed during the cycle.

This means that the stories can be "done" in one sprint, and the sprint ends with code that can be released.

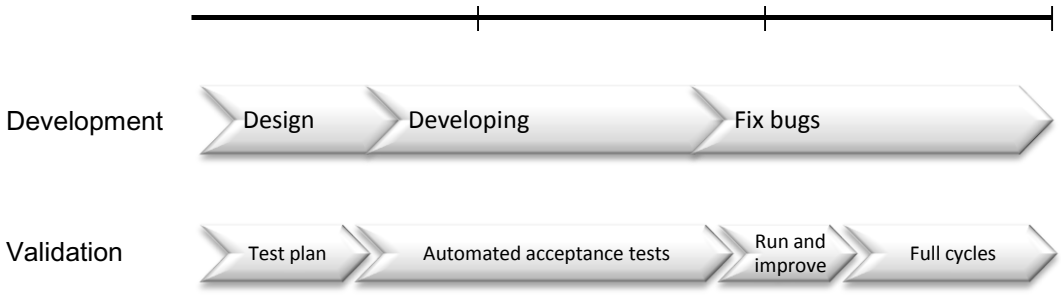


Figure 6: Three Week Sprint with ATDD

5.4 Automatic Regression

The automatic tests written as part of a sprint are re-used in the next sprints for the regression test cycles. This package of tests increases with the sprints, and makes sure that all the main flows are covered by automated tests. This lets you run many more tests in the next sprints without wasting time on manual regression testing. This greatly improves the regression test cycles. The automated regression tests can also be used for future projects.

After introducing ATDD to our process, the average number of tests run per sprint increased from 64 to 150. This increase in tests helped us find many more bugs and improve the quality (see Figure 7 and Figure 8).

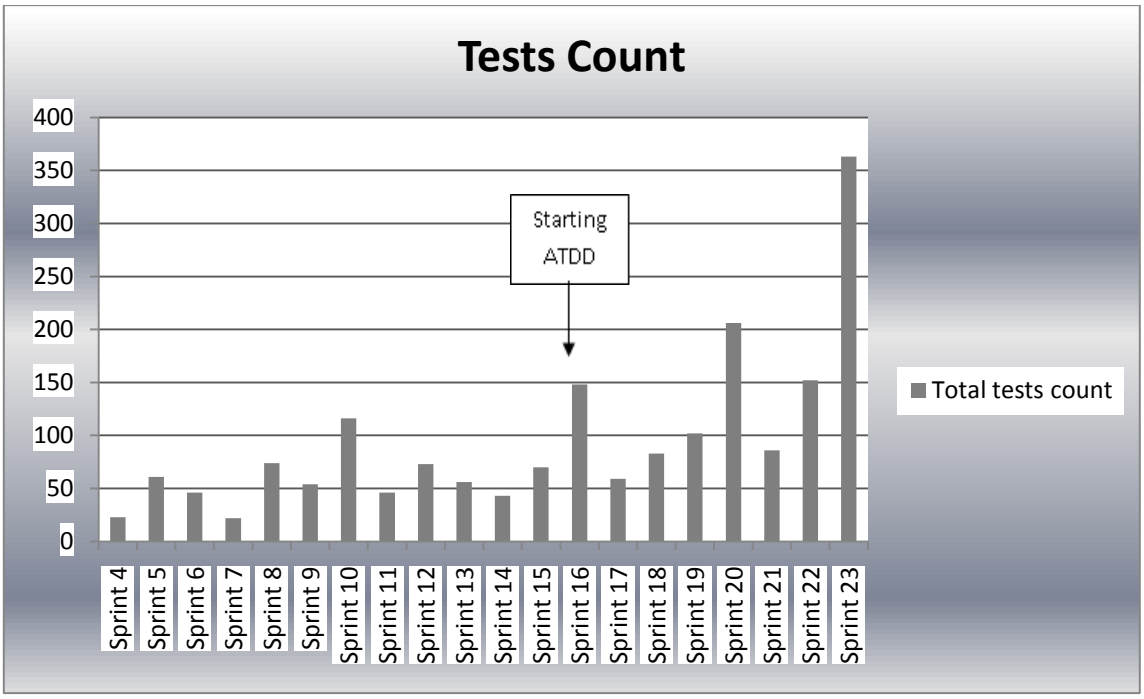


Figure 7: Tests Count

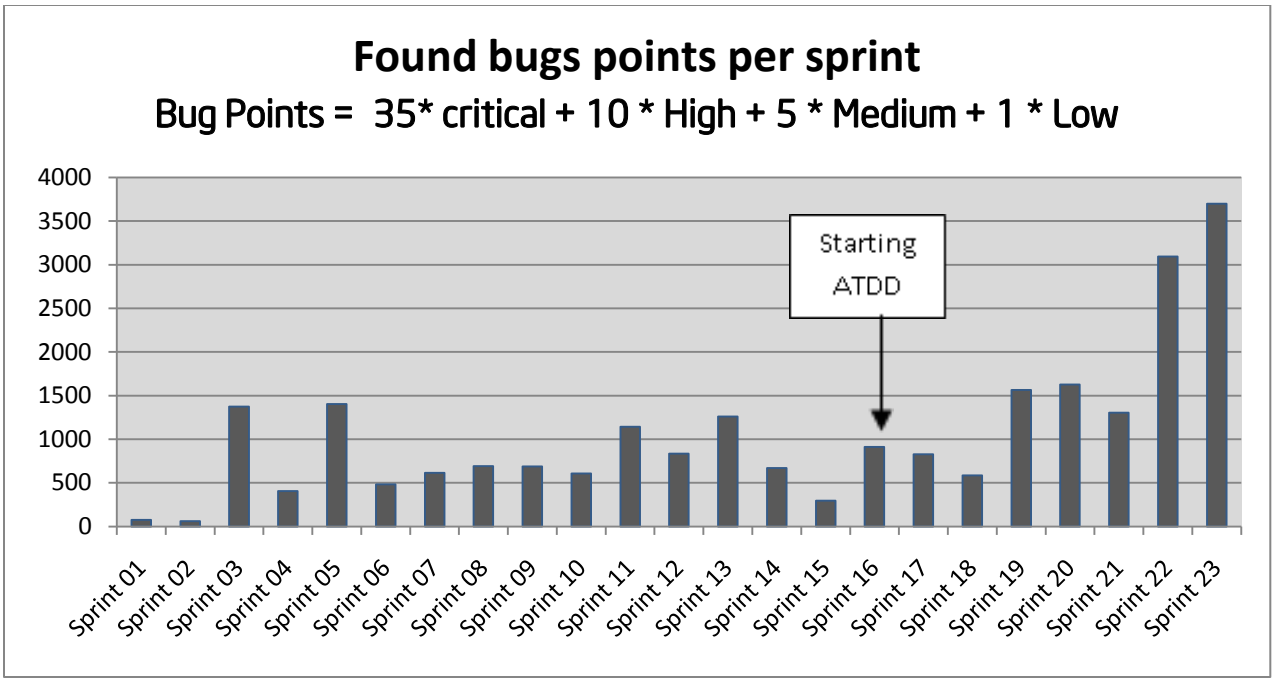


Figure 8: Bug Points Per Sprint

5.5 Quality Improvement

As a result of the improvements gained by including ATDD, and the problems that it solves, the quality of new features is improved. The testers are involved from the design phase, and the whole team is focused on the acceptance criteria.

The ease of running the regression ensures that the project is stable. Stories are completed with validation and bug fixing is all done in one sprint. Bugs are fixed very close to when they were opened. This avoids a technical debt of unresolved bugs and incomplete stories. The pass rate of the full cycles that were run after the acceptance tests passed increased from an average of 71% to 87% (see Figure 9).

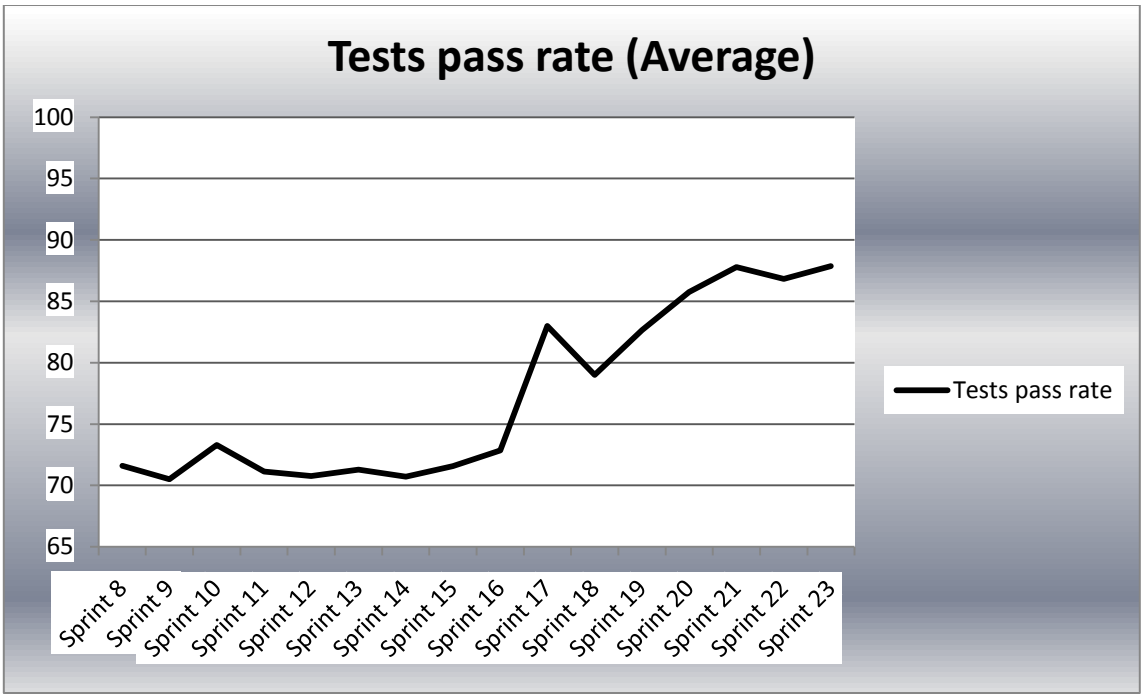


Figure 9: Tests Pass Rate Average

5.6 Team Satisfaction

The feedback that we got from the Scrum team was very good. The validation team members said it was much more fun to work this way. They enjoyed the test development process, and the test cycles became much less frustrating. Being an integral part of the sprint and the fact that bugs were fixed very close to when they were opened also improved their positive feeling. The pressure on validation at the end of the sprint was also reduced.

The main item that returns in each sprint retrospective is “Great communication between developers and validation”. The Scrum team dynamics and satisfaction improved, the information was shared quickly and efficiently, and the whole team was focused on the customer’s requirements.

6. Recommendations

6.1 Use Automation Tools

If you cannot write automated acceptance tests for your product, you cannot use ATDD. Writing automated acceptance tests requires a suitable tool for testing your particular type of product. There are ready made tools for acceptance tests, but we used our own automation tool that was built for our project.

6.2 Use Testers with Developer Skills

The testers must be able to write automated test scenarios for the acceptance criteria.
Use testers with developing skills, or teach them. They need to know at least how to use your automation tool to add the test scenarios. If the testers do not have the necessary developer skills, it will not work.

6.3 Plan for Maintenance

Automated tests need maintenance. Test automation always breaks down at some point. Bugs are found in the automation, and you will need resources to fix the bugs and to improve the structure. Make sure that you include this overhead in your plans.

6.4 Do Not use ATDD for Short Projects

For short projects, if you will not re-use the automation, do not use ATDD. It will take too much time and effort to learn the tool and to maintain the automation.

In our second short project (4 sprints), which was a subset of the first project with some changes in the usage, ATDD did not give us added value. The automation process was not stable until the end of the project. We had planned to use this automation for future projects, so it was worthwhile, but the ATDD process did not work well.

6.5 Do Not Completely Trust your Automation

Automated tests are code, and code can have bugs. Make sure you carefully review the code. Sometimes a critical bug in a project is hidden behind an automation bug.

The automated acceptance tests are good for the main functionality of your project. The aim is breadth, not depth. Use manual tests for the depth tests and let your testers' experience lead them to the important bugs. Also, different testers have different ideas of how to test things. When you trust only automated tests, you lose the new ideas and always repeat the same tests.

7. References

- **http://www.devx.com/codemag/Article/38611/1763/page/1** Introduction to Scrum, Hector J. Correa
- **http://martinfowler.com/bliki/TechnicalDebt.html** Technical Debt, Martin Fowler
- **http://www.methodsandtools.com/archive/archive.php?id=72** Acceptance TDD Explained, Lasse Koskela
- **http://www.51testing.com/ddimg/uploadsoft/20091221/51Testing_salon_41.pdf** ATDD in Practice, Xu Yi (徐毅)
- **http://www.io.com/~wazmo/succpap.htm** Success with Test Automation, Bret Pettichord

Person-to-Person Communications: Models and Applications

Richard Brenner
Chaco Canyon Consulting
www.ChacoCanyon.com
rbrenner@ChacoCanyon.com

Abstract

When we talk, listen, read or write email, or leave or listen to voicemail, we're communicating person-to-person. Whenever we do that, we take risks. We risk misunderstanding, offense, and confusion. So much can go wrong that it's impractical to learn how to fix it after the damage is done. When things do go wrong, the costs can be unbearable. Careers can founder; new products and even companies can fail.

It's far better to minimize the occurrence problems altogether. If everyone understands how communications fail, they can frame communications to avoid problems.

Attendees learn a model of communications that can help them avoid the ditches. Virginia Satir, a family therapist who applied systems thinking to human relationships, originated the model.

But we need access to what we know in those intense moments when we're most likely to slip, and least likely to remember what we've learned. That's why, in this session, we use an interactive learning model that is both effective and fun.

Of all the stressful situations we encounter at work, one of the most difficult is saying no to power. We'll show how to apply the techniques we learn to that very tricky situation.

Biography

Rick Brenner is principal of Chaco Canyon Consulting. He works with people in problem-solving organizations that need state-of-the-art teamwork and with organizations that want to create innovative products by building stronger relationships among their people. In his 25 years as a software developer, project manager, software development manager, entrepreneur and consultant, he has developed valuable insights into the interactions between people in the workplace environment, and between people and the media in which they work. He coaches managers at all levels.

Mr. Brenner has held positions in software development and software development management, at Symbolics, Inc., and at Draper Laboratory, where he conducted research into the software development process. Since 1993, he has taught a course in business modeling at the Harvard University Extension School.

Mr. Brenner holds a Masters Degree in Electrical Engineering from MIT. His current interests focus on improving personal and organizational effectiveness in abnormal situations, such as dramatic change, enterprise emergencies, and high-pressure project situations. He has written a number of essays on these subjects, available at his Web site, http://www.ChacoCanyon.com/, and writes and edits a weekly email newsletter, Point Lookout.

Person-to-Person Communications: Models and Applications

presented to the

Pacific Northwest Software Quality Conference

by

Rick Brenner
Chaco Canyon Consulting

Building State-of-the-Art Teamwork
In Problem-Solving Organizations

www.ChacoCanyon.com

1

A note on format

- Underlined items are live links to articles on my Web site or elsewhere
- To get a copy with working links, go to: http://tinyurl.com/26rs5g2
- To get a copy of the handout, go to: http://tinyurl.com/2abmgoy
- To get both as a ZIP archive: http://tinyurl.com/2f29szc

Please ask questions as we go along

2

Core Message

- Person-to-person communications are complex
  - Problems that do arise are difficult to fix
  - Prevention is easier than repair
  - We do better when we slow down
  - We have little control over what others do with what we say
- Saying "No" to power can be difficult — so we often say "Yes"
  - But we can learn how to say no…
  - …and when not to say yes

Work on preventing problems by changing our inner processes

3

Learning is the art of acquiring new ideas

- Whatever you've been doing is the best you know how
- Honor what you know as good
- You might someday learn something better
- Your toolbox isn't full—you can add new things without discarding the old

4

The Communication Process

Person-to-Person Interactions

5

What are communications?

- A process in which information is *exchanged*
- Requires a common system of tokens: symbols, signs or behavior
  - Pheromones in insects
  - Words, expressions, body language in humans

When we do well, we communicate *with* each other.

6

Examples of problem communications

- Living the catastrophic expectation
- Implied accusations
- The Tweaking CC
- Commitment by implication
- Mind-reading
- Culture/gender/generational differences
- Hat-hanging
- Mistakes

7

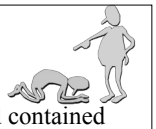
Living the catastrophic expectation

- Receive an email query "how did we decide this?"
- Respond with defense of the outcome
- Problem:
  - The query asked only about the decision process
  - The response went immediately to defending an attack on the decision

Living the catastrophic expectation can shift us into a high-stress mode unnecessarily

8

Implied accusations



- An accusation presumed as true and contained within another statement
- Examples:
  - I will agree to invite you to the meeting, if you will agree to be polite and not interrupt other people
  - You can join the team if you agree not to pout if your ideas are not accepted

Implied accusations can put your communication partner on the defensive

9

The Tweaking CC

- The tweaking CC is a pressure tactic
- A CC to the recipient's supervisor or other powerful people
- Message usually contains embarrassing information

Pressuring your communication partner rarely has the intended effect

10

Commitment by implication

- Someone else rephrases your words so as to commit you to something you didn't intend
- Example:
  - You said: "Yes, I'll get you a few names—not many—soon."
  - Response: "Great! I'll take quality over quantity any time!"

Commitments are real only if they are made with freedom

11

Mind-reading

- None of us can *really* read minds
- We just *think* we can
  - "You're only saying that because you want me to agree with you"
  - "I don't believe that because you would never do that unless there was more in it for you"
- Conclusions based on mind-reading are suspect

Mind-reading is so pervasive that we no longer recognize it as such

12

Culture/gender/generational differences

- We all use similar communication forms
- Different cultures and sexes attach different meanings to the same forms
  - "Would you like to"
  - Politeness/wimpiness

When we assume that communication tokens have universal meanings, we're headed for trouble

13

Hat-hanging

- People we meet sometimes match up with people we knew long ago (parents, old bosses, etc.)
- We tend to attribute characteristics of people we knew then to people we know now
- Basis of advertising and con games

Hat-hanging is very difficult to detect and almost always a disaster

14

Mistakes

- We're all imperfect
- We make "simple" mistakes, especially under stress
- When our communication partners are also under stress:
  - They interpret the mistake on as-if-intended basis
  - This is also a mistake

Mistakes, especially under stress, can damage a relationship irreparably

15

What we learn from all these problems

- Person-to-person communication is riddled with problems
- The possibilities for defects seem endless
- Fixing defects in this system is a lot harder than fixing a production process
- To improve quality, work at a deep level

A case-by-case approach is unlikely to make a measurable difference—too many cases

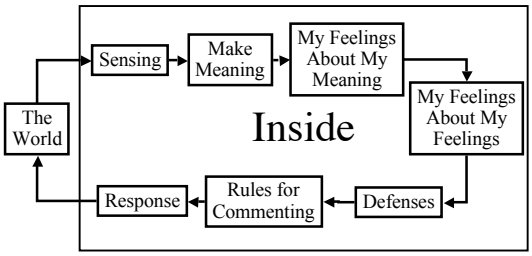
16

A model of person-to-person interaction

- When we converse:
  you talk, I talk, you talk, etc.
- But it's way more complex than that
- Things happen inside us
- To improve effectiveness of communication:
  - Create a system model of the inside and the outside
  - Use the model to understand defects
  - Devise approaches to limit the occurrence of defects

17

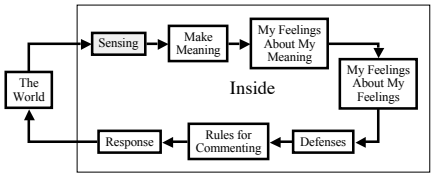
A model of human interaction



18

Example: Sensing

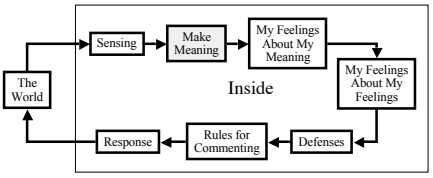
- Taking in data through the senses
- "She fidgeted as she spoke, and backtracked several times."



19

Example: Make meaning

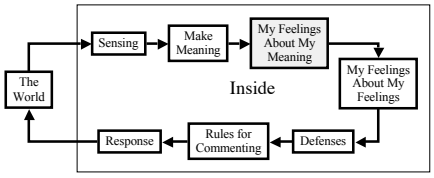
- Making meaning of the data we take in
- "She seemed nervous and unsure of what she was talking about"



20

Example: My feelings about my meaning

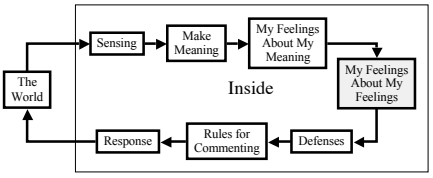
- How we feel about the meaning we made from the data
- "I feel bad for her and want to help her if I can."



21

Example: My feelings about my feelings

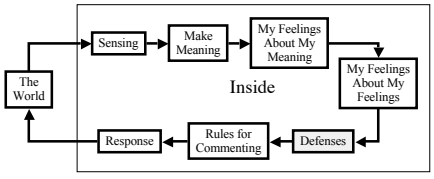
- How we feel about having those feelings
- "I shouldn't want to help her. I feel guilty about wanting to."



22

Example: My defenses

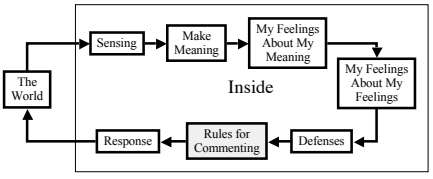
- Protection from our own feelings"I'm angry about being sympathetic. There will be none of that."



23

Example: My rules for commenting

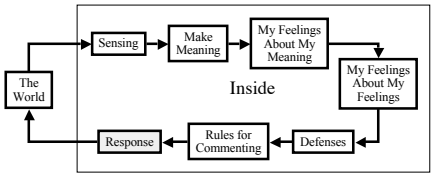
- Rules governing disclosure to others
- "I must not let anyone know that I feel sympathy. I must be tough."



24

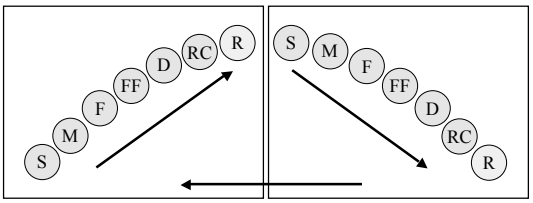
Example: My response

- What we do about what we took in, subject to the constraints of our rules and defenses
- "Pull yourself together!"



25

Exercise: Interaction model



26

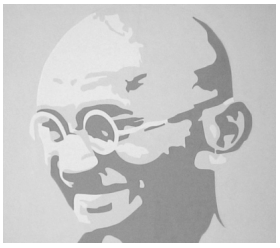
Homework: Interaction model

- Find an interaction at work that stirred up emotions
- Try to reconstruct the part of the interchange that got hot
- Play out the interaction model—what happened in each step?

27

Saying No

28



A 'No' uttered from deepest conviction
is better and greater than
a 'Yes' merely uttered to please,
or what is worse, to avoid trouble.

—Mohandas Gandhi

29

Saying No: the basics

- Saying "No" to power can be difficult — so we often say "Yes"
- Agreeing to the unworkable doesn't work
- Learn why saying no is hard for ***you***
- Learn techniques for saying no
- For best results:

Move the discussion away from confrontation toward problem solving

30

How I relearned about saying no

31

When is saying no difficult?

- When pleasing others is a primary goal
- When we believe we should say yes
- When we wish things were otherwise
- When other people want us to say yes
- When other people control things we want
- When other people trigger our emotions

The more tempting it is to say "Yes" the more difficult it is to say "No"

32

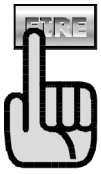
Example "no" scenarios

- Work life
  - Setting a meeting agenda
  - Inviting people to a meeting
- Project management
  - Slipping a date
  - Reducing resources
  - Changing requirements
  - Negotiating targets
- Everyday life
  - Introducing yourself
  - Checkout counter

33

The fundamental problem: Internal forces take over

- When our self-image is on the line
- When we don't see how to protect ourselves from consequences
- When others push our buttons



Central issue: how to cope with discomfort

35

How to understand the dynamics

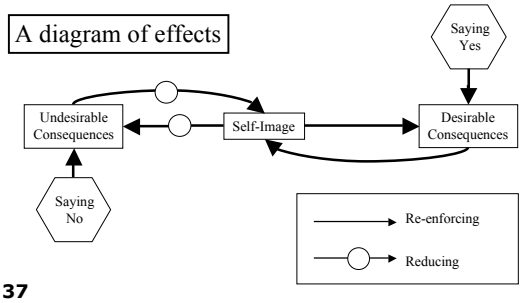
Understand:

- How self-image is involved
- How saying yes *seems* to protect self-image
- Partner's role

36

A *simplistic* view of the yes/no dynamic

A diagram of effects



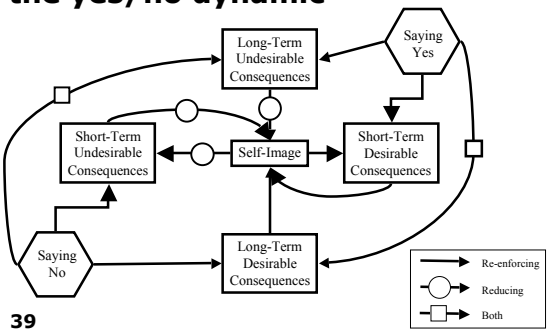
37

To learn to say no

- Adopt a more realistic view of the yes/no dynamic
  - Distinguish long-term and short-term consequences
  - Allow for both "positive" and "negative" effects of your actions
- Learn how to move the group toward problem solving

38

A *more realistic* view of the yes/no dynamic



39

Where the pressure comes from

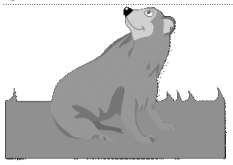
- Solving a problem with the wrong tool
- We invoke (nonexistent) engineering magic to:
  - Solve a financial problem
  - Solve a marketing problem
  - Conceal a strategic error
  - Cover over or address a political problem
- Solving a problem with the wrong tool is usually a doomed effort

Pressure is greatest when project teams have least political power

40

What can go wrong when we say yes but we want to say no

- Saying Yes doesn't make the impossible possible
- Undermines your credibility with your team
- When you can't deliver, you lose credibility with your pressuring partner
- Ground Hog Day
- Team burns out
- Work quality degrades
- Time passes
- Problems multiply



41

What about <u>ethics</u>?

- Saying yes introduces delay
  - Delay the confrontation with your pressuring partner
  - Delay the confrontation with reality
- Delay can foreclose alternative(s)
- We say we're "buying time"
- We're <u>actually embezzling time</u>:
  - We transfer time *from* the project's account
  - We transfer time *to* our own personal account

42

Unintended consequences of saying yes

- We help people to continue to live a fantasy
- We deprive the group of the need to address the ***real*** problem
- We miss a chance to move toward problem solving

To move the group toward problem solving, *someone* must say no

43

Tactics to avoid

- Telegraphing ahead
- The cavalier "no"
- I told you so
- Putting the punch line last
- Pulling the punch
- We're not to blame — they are
- Saying no when "no" is unacceptable

44

Tactics for saying no directly

- Start by feeling good about yourself
- Use ***I*** statements or ***We*** statements
- I don't know how to do that
- I don't know how to do that within these constraints
- I can't do that by then. Help me adjust priorities.
- Our supplier can't deliver what we need in time. Can you help with that?

45

More tactics for saying no directly

- What would happen if we were X weeks later?
- What if we add that to a later release instead?
- We could do that, but it will take X.
- We might be able to do that.
  I'd put the probability under X%.
- What can we tell them in the meantime?

46

Preparing to say no

- Know your tactics — they are your tools
- Practice with a live partner
- Expect that you won't succeed every time
- Keep a tactics collection; swap with colleagues
- Create contracts in advance
  - Schedule contract
  - Requirements contract
  - etc.

It takes two to say no –
one to say it and another to hear it

47

Homework

- Observe the people around you as they say no
- Form a say-no circle
  - Exchange say-no tactics
  - Exchange info about pressure tactics
- Sponsors: give recognition to those who say no

48

Adm. Leighton Smith, Jr. (ret.)

49

Final Word

- Much communication training emphasizes repair (e.g., conflict resolution)
- The real payoff is in damage avoidance
- We cannot control
  - What others communicate
  - What others do with what we say/do
- We *can* control
  - What *we* do
  - What *we* communicate

50

Subscribe to my free newsletter: *Point Lookout*

- Weekly newsletter by email or RSS
- 500 words per edition
- Free
- Topics:
  - Communications
  - Conflict
  - Project management
- http://www.ChacoCanyon.com/pointlookout
- http://www.ChacoCanyon.com/rss/feed.xml

Or: Write "subscribe" on your business card

51

Resources for you

- More at http://www.ChacoCanyon.com/essays/sayingno.shtml
- Links collection: http://www.ChacoCanyon.com/resources/peopleatwork.shtml

52

The Good, Bad, and the Puzzling: The Agile Experience at Five Companies

Michael Mah, QSM Associates Inc.

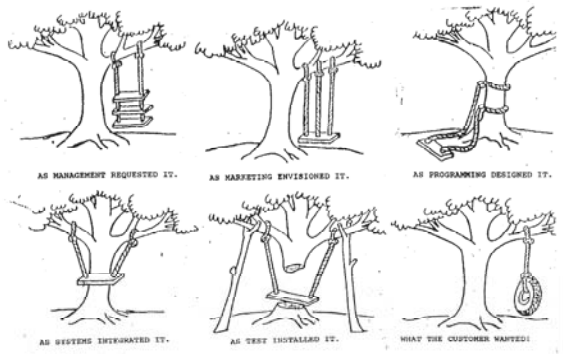
Strategic software developments – and failures – happen every day; Agile methods offer a major shift. But are they working? Drawing from industry statistics, Michael answers vital questions about Agile's effectiveness, which may be turning the "law of software physics" upside down. Until now, there have been predictable relationships among schedule, staffing, and quality; industry data indicates Agile may be changing all this. See productivity findings at five Agile companies, and the results for time-to-market, productivity, and quality. Learn the right practices for your environment, including characteristics of successful measurement. See how metrics reveal insights into Agile approaches that are becoming mainstream.

Michael Mah *is managing partner at QSM Associates Inc. He teaches, writes, and consults to technology companies on estimating and managing software projects, whether in-house, offshore, waterfall, or agile. He is the director of the Benchmarking Practice at the Cutter Consortium, a US-based IT think-tank. With over 25 years of experience, QSM has derived productivity patterns for thousands of projects collected worldwide. His work examines time-pressure dynamics of teams, and its role in project success and failure. His background is in physics and electrical engineering, and he is a mediator specializing in conflict resolution for technology projects.*

Website: www.qsma.com.
Email: Michael.mah@qsma.com
Blog: www.optimalfriction.com
Phone: 413-499-0988

QSM ASSOCIATES | Project Management That Learns From Experience

QSM ASSOCIATES | Project Management That Learns From Experience



QSM ASSOCIATES | Project Management That Learns From Experience

"Frothy eloquence neither convinces nor satisfies me. I am from Missouri. You have got to show me."

- Missouri Congressman Willard Duncan Vandiver, 1899

QSM ASSOCIATES | Project Management That Learns From Experience

Co-Located XP - Follett Software

- Team size
  - 24 Developers
  - 7 Testers
  - 3 Customers
  - 3 Project Leaders
- Code Base
  - 1,000,000 lines of code
  - 7,000 automated unit test
  - 10,000 automated acceptance test

QSM ASSOCIATES | Project Management That Learns From Experience

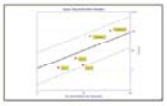
Agile Release Retrospectives

1 - COLLECT AND VALIDATE PROJECT DATA

SLIM Metrics

2 - ANALYZE PROJECTS USING QSM REFERENCE DATABASE

4 - PRESENT RESULTS



3 - DETERMINE PROCESS METRICS & PROJECT POSITIONING

QSM ASSOCIATES | Project Management That Learns From Experience

Four Core Metrics: *Your History*

Produced Software (Size)

How long? → Duration

How much? → Effort

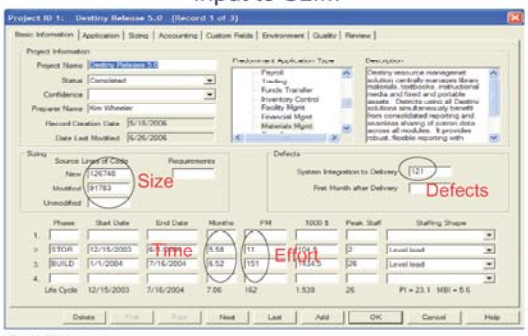
How good? → Discovered Defects

Productivity - "The Fifth Metric"

QSM ASSOCIATES

Project Management
That Learns From Experience

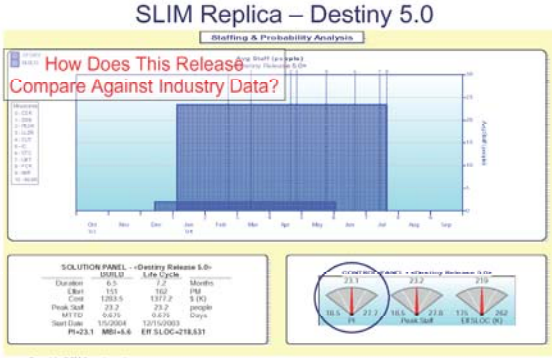
Input to SLIM



QSM ASSOCIATES

Project Management
That Learns From Experience

SLIM Replica – Destiny 5.0



QSM ASSOCIATES

Project Management
That Learns From Experience

Follett vs. Industry Average

	Industry Average	Current Performance	Delta
Project Cost	$3.5 Million	$2.2 Million	-$1.3M
Schedule	12.6 months	7.8 months	-4.8 mos
QA Defects	242	121	-50%
Staffing	35	35	n/a

QSM ASSOCIATES

Project Management
That Learns From Experience

Offshore vs. Industry Average

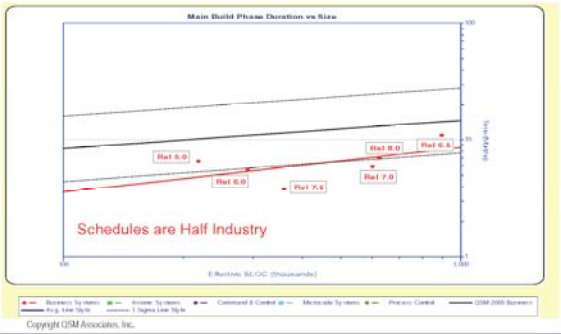
	Industry Average	Offshore Average	Delta
Project Cost	$3.5 Million	$3.2 Million	-$0.3M
Schedule	12.6 months	9.6 months	-3.0 mos
QA Defects	242	677	+280%
Staffing	35	50	+15

* Normalized to same project size

QSM ASSOCIATES

Project Management
That Learns From Experience

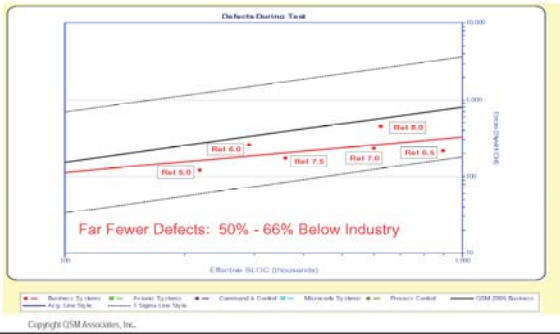
Trendline Assessment – Build Phase Schedule



QSM ASSOCIATES

Project Management
That Learns From Experience

Trendline Assessment – Defects/Quality



QSM ASSOCIATES | Project Management That Learns From Experience

Follett and XP: It has worked incredibly well…

- Destiny Library Manager:
  - Award of Excellence 2004, presented by Technology and Learning magazine (December 2004).
  - Awards Portfolio 2004, presented by Media and Methods magazine (May/June 2004).
  - Technology & Learning Award of Excellence 2006, 2007
- Destiny Textbook Manager
  - Awards Portfolio 2005, presented by Media and Methods magazine (May/June 2005).
  - Technology & Learning Award of Excellence 2007
- Destiny Enriched Services
  - Technology & Learning Award of Excellence 2007

Follett Software provides Library Automation Solutions to *52% of the K12 market*. Destiny Library Manager: *Single largest product market share* in K12 with 19% of the total market and continues to *outpace the competition in market growth*.

QSM ASSOCIATES | Project Management That Learns From Experience

Domain Knowledge

- Smart people, experienced people
- Coding is moving knowledge from mind into the machine
- Inexperience costs money

QSM ASSOCIATES | Project Management That Learns From Experience

Short Feedback Loops

- Paired programmers
- Instantaneous code reviews
- Accelerated learning and execution
- Face to face communication channel

QSM ASSOCIATES | Project Management That Learns From Experience

Time Boxing

- Short iterations
- Clear and discernible progress
- Anticipation of the next important feature
- Efficiency

QSM ASSOCIATES | Project Management That Learns From Experience

Avoiding Burnout

- XP = Sustainable pace
- 40 Hour Work Weeks
- Prevent productivity collapse for overworked teams

QSM ASSOCIATES | Project Management That Learns From Experience

Craftsmanship Over Craft

- Take pride in what you do
- Do not compromise professionalism
- Simple design
- Upfront testing
- Prevent costly rework
- Build it right the first time
- "Measure Twice, Cut Once" (says Norm Abram)

QSM ASSOCIATES

Project Management
That Learns From Experience

Transparency

- The best teams hide nothing
- Release Planning
- Daily stand-ups/scrum sessions
- Iteration Demos
- Velocity and burn-down charts

QSM ASSOCIATES

Project Management
That Learns From Experience

High-bandwidth Communication

- The best teams have "wide-open pipes"
- Domain knowledge moves among the team
- Information flows rapidly and accurately

QSM ASSOCIATES

Project Management
That Learns From Experience

Avoiding Waste and Costly Rework

- Rework has high cost
- Rework takes time
- Rework creates defects
- Rework is bad
- Refactoring can be a cover up

QSM ASSOCIATES

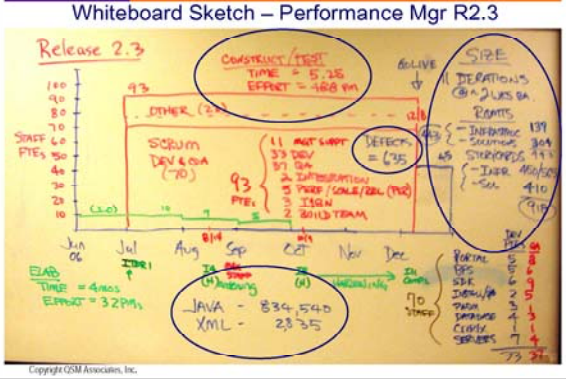
Project Management
That Learns From Experience

Distributed SCRUM – BMC Software

QSM ASSOCIATES

Project Management
That Learns From Experience

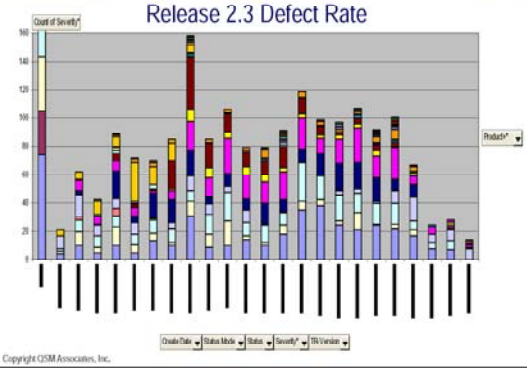
Whiteboard Sketch – Performance Mgr R2.3



QSM ASSOCIATES

Project Management
That Learns From Experience

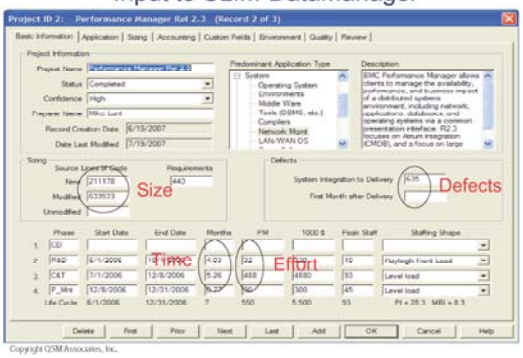
Release 2.3 Defect Rate



QSM ASSOCIATES

Project Management
That Learns From Experience

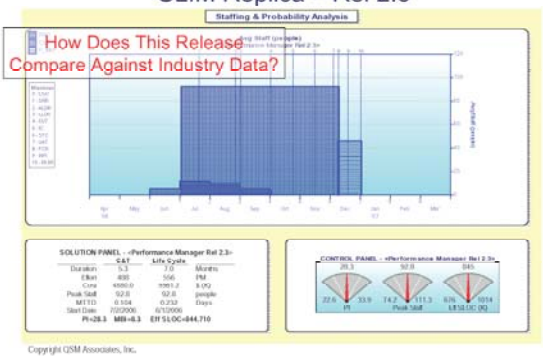
Input to SLIM-DataManager



QSM ASSOCIATES

Project Management
That Learns From Experience

SLIM Replica – Rel 2.3



QSM ASSOCIATES

Project Management
That Learns From Experience

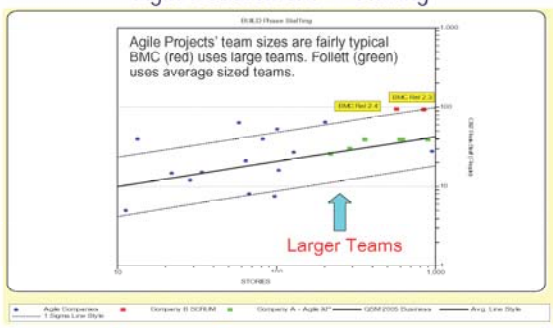
BMC vs. Industry Average

	Industry Average	Current Performance	Delta
Project Cost	$5.5 Million	$5.2 Million	-$.3M
Schedule	15 months	6.3 months	-8.7 mos
QA Defects	713	635	-11%
Staffing	40	92	+52

QSM ASSOCIATES

Project Management
That Learns From Experience

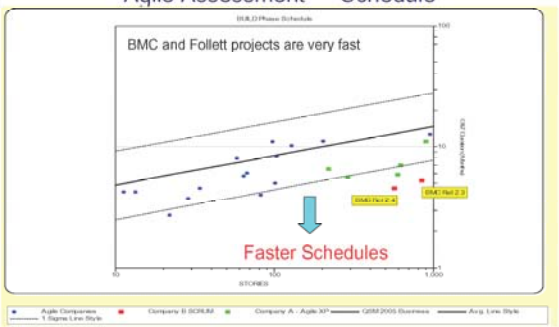
Agile Assessment — Staffing



QSM ASSOCIATES

Project Management
That Learns From Experience

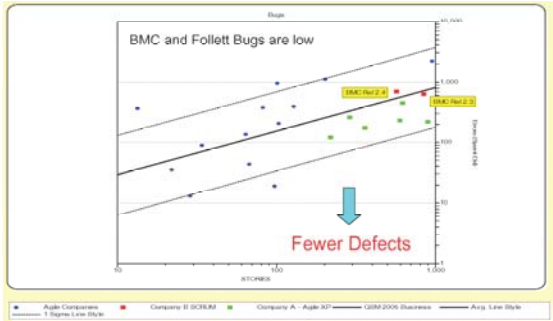
Agile Assessment — Schedule



QSM ASSOCIATES

Project Management
That Learns From Experience

Agile Assessment – Quality



QSM ASSOCIATES | Project Management That Learns From Experience

BMC "Secret Sauce"

QSM ASSOCIATES | Project Management That Learns From Experience

BMC "Secret Sauce" (con't)

- Buy-In
  - VP-Level (or higher) Senior Executive Sponsorship
  - Scrum Master Training
  - Core Group Energized and Passionate
- Staying "Releasable"
  - Nightly Builds/Test
  - 2-week Iteration Demos
  - Frequent, Rigorous Peer Code Review
- Dusk-to-Dawn Teamwork
  - Communication Techniques for Information Flow
  - Wikis, Video-conferencing, Periodic On-Site Meetings
  - Co-Located Release Planning
  - Scrum of Scrum Meetings (US Time)

QSM ASSOCIATES | Project Management That Learns From Experience

BMC "Secret Sauce" (con't)

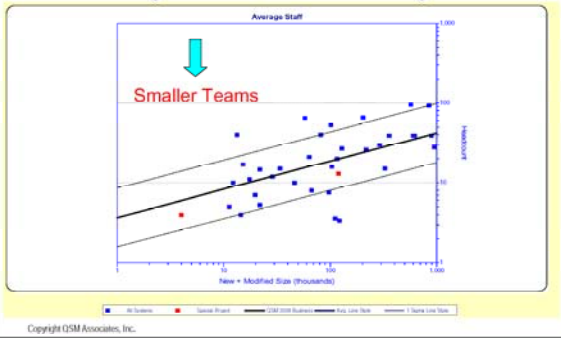
- Backlogs
  - One Master Backlog AND Multiple Backlog Management
  - One Setup for User Stories Across Teams
  - Added "Requirements Architect" to Interface Product Mgt with R&D
- "Holding Back the Waterfall"
  - Test Driven Development
  - Retrospective Meetings to Not Regress into old Waterfall Habits
  - Outside Source to Audit the Process

QSM ASSOCIATES | Project Management That Learns From Experience

Agile at Quick Solutions

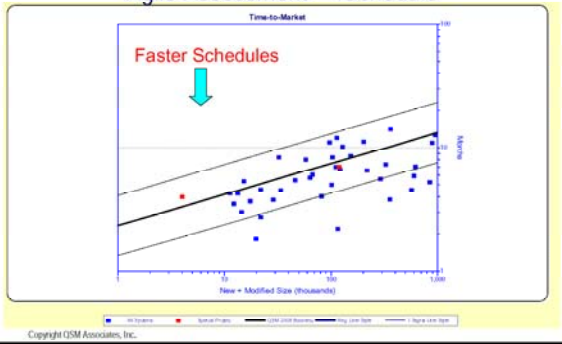
QSM ASSOCIATES | Project Management That Learns From Experience

Agile Assessment — Staffing



QSM ASSOCIATES | Project Management That Learns From Experience

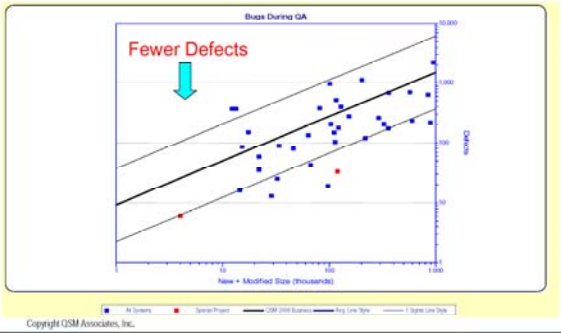
Agile Assessment — Schedule



QSM ASSOCIATES

Project Management
That Learns From Experience

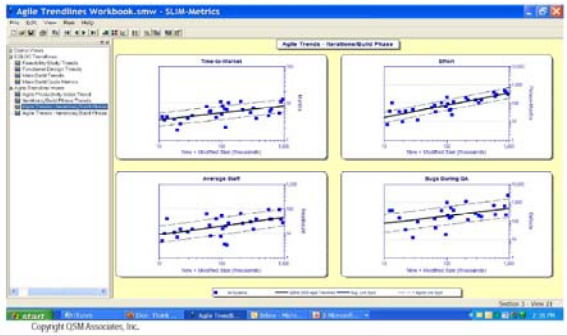
Agile Assessment – Quality



QSM ASSOCIATES

Project Management
That Learns From Experience

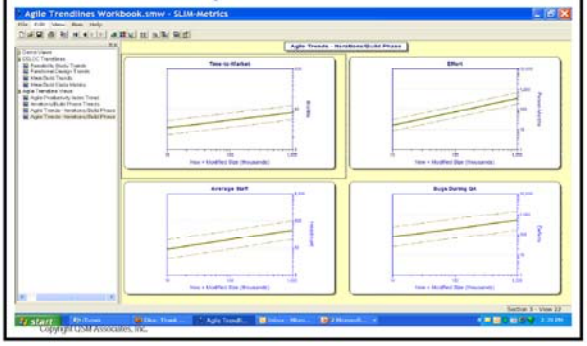
Agile Database Creates Agile Trends



QSM ASSOCIATES

Project Management
That Learns From Experience

New Agile Benchmark Trends



QSM ASSOCIATES

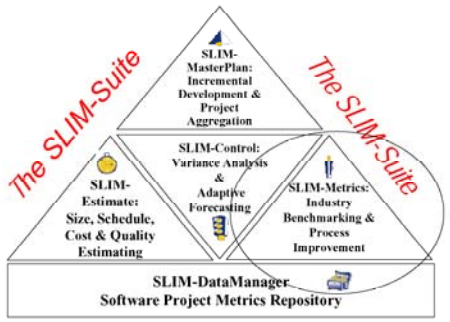
Project Management
That Learns From Experience

Why is Agile So Important?

- Cause SOFTWARE is so important
- It is TOO important to screw it up
- Software is difficult, because it is BIGGER than any one of us. It requires a collective of human minds to build today's modern software applications
- Some consider software to be the wiring of the Global Digital Nervous System of the Earth, part of wiring the Global Brain

QSM ASSOCIATES

Project Management
That Learns From Experience



Don’t test too much! [or too little]
(Lessons learned the hard way)

Keith Stobie
Keith.Stobie@microsoft.com

Abstract

Over-testing can actually be as bad for you as under-testing. Whether it is testing to your own lofty expectations (versus those of the project), verifying too much in stress, or verifying too much in one test, you might end up with a less useful result than other approaches. You must also be careful of under-testing sequence and state or aspects of the software that are just hard to verify.

This experience report relates parables of mistakes I made and what I learned in the process, so you can avoid these mistakes. These lessons learned are also cross indexed to the book: *Lessons Learned in Software Testing* (Kaner, Bach, and Pettichord).

Biography

Keith Stobie is a Test Architect for Bing Infrastructure at Microsoft where he plans, designs, and reviews software architecture and tests. Previously, 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

The many times I've taught various aspects of software testing, I've found many students learn best by hearing stories that demonstrate a principle in action. During my years of testing applications I have gathered a number of scars that I would prefer newcomers to the discipline avoid. Many of my lessons learned occurred before a book called *Lessons Learned in Software Testing* [Kaner 2001] was ever published. But in reading the book I could see how my experiences related to those in the book and correlate them as lesson #. The majority of the lessons are a direct result of my mistakes around over-testing!

2 Over- and Under-testing Experiences

What is over-testing? How can you test too much? These are typically quandaries from new testers who don't understand the business context in which most testing is done. Many books on testing state that testing is an infinite task. Business drivers dictate finite resources; the choice of what to test and how extensively is critical. Over-testing one area means you have used resources that may be better spent testing other areas.

I have learned many testing techniques over the years and have a catalog of over 50 different analysis methods. Why do we need so many techniques and analysis methods? Because software rarely shows its quality through only one aspect. The principle from Kaner's book that I most believe in is lesson #283, "Apply diverse half measures". This is also illustrated in the Manager's Guide to Evaluating Test Suites [Marick 2000], reporting on a very defect-prone portion while ignoring the rest of the product.

2.1 Verify Stress Tests Out of Band

A major difference between functional and stress tests is that stress tests typically run in a constrained resource environment. Stress wants as much (or even too much) work done as the system is capable of in the least amount of time, by consuming all of the CPU cycles, all of the I/O bandwidth, etc. Running functional tests in a loop has been shown not to be the most effective stress test.

Many systems must have much larger test driver systems than the actual system under test. For example, they might need 4 client machines to keep 1 server machine busy. Frequently the driver tests are not coded for optimal performance. Driver tests may cause unnecessary and useless work, for example driving a browser UI to generate HTTP traffic for a server. Driver tests may also make redundant or unnecessary verifications and thus they may check too much.

One of the earliest, most dramatic examples of this was when I tested a transaction system. Transaction systems let multiple concurrent users, applications, or threads, simultaneously access a resource, canonically a database system, with each getting their own consistent view of the resource. To do this, a Transaction system maintains Atomicity, Consistency, Isolation, and Durability, or ACID properties, of the results. I wrote a test system a model as it were - that mirrored the logical result of each operation, so that I could tell the transaction system did the same thing as my model. This worked great for functional testing. However, my model operated much slower than the real system, and thus concurrency was actually quite limited when used in a stress test.

Another engineer chose a much more clever approach. To measure Isolation, each database change was tagged with the test instance making the change. To measure Atomicity and Consistency the changes within a transaction were deliberately inconsistent with only the ending result restoring consistency. For example, in an ATM transaction when transferring money from Savings to Checking, you have two operations within a transaction. One operation debits savings and the other operation credits checking, but either both operations or neither (atomicity) must complete to keep consistency.

We could include a second set of operations within the same transaction, e.g. transferring money from Checking to Savings. Now, if we transferred the same amount both ways, we could not tell if all the

operations or none of the operations succeeded. However, by checking a consistency property of the result, we can. Our test decides it will maintain all account balances as evenly divisible by 1,000. Suppose Checking and Savings both start with 10,000 as their balance. For the first transfer, we can randomly choose any amount to transfer, e.g. 1,234.57. However this would leave our balances not evenly divisible by 1,000. We create a consistency-preserving second transfer. It could transfer more money, e.g.

First Savings -> Checking 1,234.57
Second Savings -> Checking 765.43

The completion of this transaction leaves savings 8,000 and checking at 12,000 which are both evenly divisible by 1,000, but if any 1, 2 or 3 operations within the transaction were not complete, then consistency was lost. Instead of a second addition, we could have done a subtraction, e.g.

First Savings -> Checking 1,234.57
Second Checking -> Savings 234.57

The complete of this transaction leaves savings at 9,000 and checking at 11,000, both evenly divisible by 1000. Again, if any 1, 2 or 3 operations within the transaction were not complete, then consistency was lost.

Now independent operations could randomly build sets of operations that each measured consistency individually. The tests could run as many operations as fast as the system could handle for as long as you wanted to test. At the end of the test, a simple check would tell you if consistency (and atomicity and Isolation) had been maintained. Diagnosing the failure was far more difficult. If consistency was lost, you would have to use the data logged by the tests into the test data for clues. You could also use the logs kept by the transaction system and its notion of roll-back to isolate when the failure occurred.

The inability to pinpoint the time and location allowed for much greater concurrency with limited resources. The first several times this approach was run numerous race conditions never exposed by the earlier method, which checked results in real time, were exposed.

Lesson – my original functional transaction validator validated too well. It assumed too much knowledge which limited its speed while consuming great resources. A simpler, out-of-band verification worked much better.

2.2 Test Oracle Complexity <= System Under Test Complexity

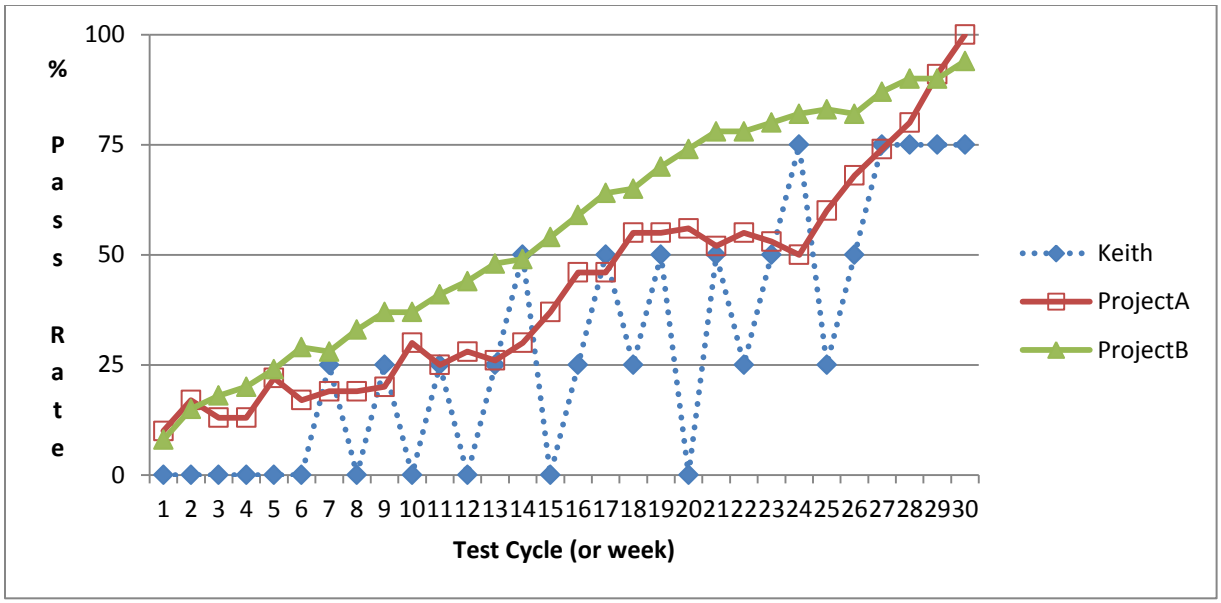
Another rule I've learned, which this example illustrates, is your test oracle for determining correct results should never be more complicated than the system you are testing. Test oracles are just as error-prone (if not more error-prone) as a system under test. The worst case test oracle should be doing what is called n-version programming. Create two (or more) versions of a system and compare the results. There are numerous papers on this approach and its limitations (e.g. both implementations sharing bugs from a buggy specification document). My simplest example of this is testing the Square Root function. Test shouldn't create another square root function to compare. Test should use a simpler oracle, which is just to square the result and compare with the original!

Related ideas for partial test oracles can be found under topics like metamorphic testing [Chen 2004, Hu 2006], sampling oracles [Hoffman 1998], or consistency oracles [Hoffman 2006].

2.3 Test in the Presence of Failures

I have never shipped a major piece of software defect free. Rather than explaining why it is OK to ship with bugs, I will concentrate on the testing implications of the fact.

I have one development manager who will never forgive me for the grief I caused their project. I wrote only four rather large "test cases". These four test cases covered all of the functionality of the project. However, the company expected teams to report percentage of test cases passing (as though that is a useful number). While most teams showed pretty wavy lines typically growing towards 90% or better passing rate, my development manager had a nasty step function, for example the dashed line below:



Part of the problem is using the amorphous, ill-defined, "test case" as a unit of measure (lesson #280 "How to lie with test cases"). I've seen test cases vary from 5 lines of text or code to hundreds of pages of text and thousands of lines of code. A better measure relates to number of requirements or features shown to be working (lesson #34 "It Works" really means it appears to meet some requirement to some degree.)

The other problem is each of my test cases was potentially testing too much. When I came to Microsoft and started using formal Model Based Testing tools this became most starkly evident. Research liked the idea of generating a single test case that covered everything in a model. That is, they would generate only one test case to verify everything! On the surface this seemed great to them; If this test case passes, you know your product has no detected bugs and if it fails, you still have bugs to remove. You can get this behavior today using the "long test" strategy with Spec Explorer for Visual Studio which attempts to generate a single test to cover all transitions in the model.

The danger of my four test cases or auto-generating a single all-encompassing test case is they test too much. The simplest way to design test cases, like reliable software, is to fail fast. As soon as an error is detected, note the error and stop any further processing. Most unit testing frameworks like xUnit (http://en.wikipedia.org/wiki/XUnit) or MSTest (http://en.wikipedia.org/wiki/MSTest) incorporate this notion into their Assert statements. When an Assert fails, an exception is raised and reported. The exception stops further progress. Combining fail fast testing with only a few large test cases is a risky combination. Once a test case stops due to failure, you lose any information about other work the test case might have performed. For example, suppose a test case has 100 asserts. If the tenth assert fails, you do not know the result of the remaining 90 asserts. The failure blocks your ability to learn more information. You can work around this by changing the test case (expecting the current behavior, commenting out the offending assert, etc.), but now you have two test cases: the original and the modified.

Thus, one way to find out if you are testing too much in test cases is if you have too many blocking failures. There are numerous other reasons for not testing too much in a single test case. Test Prioritization tools work best in minimizing testing amongst many test cases that don't overlap too much. Automatic Failure Analysis tools work better when test cases are likely to find different failures instead of the same failure multiple ways. With multiple test cases (instead of one), you can run more tests in parallel using more hardware and potentially reduce your overall testing time.

Note, that I am also not a proponent of very small test cases (other than for unit testing). I like Marick's advice:

> "If you blend ideas into a more complex test, you stand a chance of inadvertently implementing a test idea that finds a bug." [Marick 2000]

and Kaner's "Appropriately complex" [Kaner 2003] concept. Choosing a reasonable test case size and complexity is important and difficult. Unit tests should be very small. Once the unit tests and perhaps basic functional tests have shown the software sufficiently stable, moderately complex tests should be used. Very complex tests, while sometimes useful, are usually difficult to maintain and to diagnose software failures from.

2.4 You Must Understand the Goals of Your Release

One of my areas of expertise is distributed transaction systems. I first did testing for Tandem's distributed transaction manager. The test suite became one of my examples of "test driven" development. When Tandem decided to rewrite the transaction system when moving from 16-bit to a 32-bit system, the developers found the documentation lacking and the code too confusing. They used the test suite as their oracle of what to build. They assumed that if they built a 32-bit version of the system with the same test results as the 16-bit system, then customers should see no difference when moving their code to the new system – and they didn't!.

I then worked on a distributed database with transactions at Informix. Finally, I was hired by BEA Systems which focuses on transactions for databases. BEA had purchased the industry standard Tuxedo transaction system [Andrade 1996] used in many vendors benchmarks. BEA was creating a Java Transactions Application Programming Interfaces (APIs) which became part of Java Second Enterprise Edition (J2EE). In designing the testing of the system, we naturally wanted to assure all of the ACID properties. With only a couple weeks before our first Beta release to customers, I and the Development Manager were shocked and discouraged to learn that the Java transaction system did not provide isolation or consistency when more than one thread was in use! We thought we had a disaster on our hands. As lesson #12 says, "Never be the gate keeper". We presented our findings to the project manager and asked if we needed to slip the release. While the project manager was discouraged by the findings, he had a better grasps of the goal of the release. We already knew the underlying Tuxedo system was robust and reliable. We were also confident that the coding flaws were not deep design flaws in our new Java interface. The purpose of the Beta was not to go into production, but to gather feedback from real users about the usability of the new Java Transactions API (JTA). Concurrent usage was not necessary to evaluate the usability of the APIs for programmers (i.e. programmers just code and execute one thread. They couldn't run multiple copies of the thread or different threads). By simply noting in the release notes that the Beta did not support concurrent users, the Beta could still achieve its primary purpose of gathering feedback on the APIs. At the same time, while Beta customers were getting used to a new API they had never seen before (this was the early days of Java), we could make the code fixes and verify concurrent usage.

The release of a product is a business decision. A test group provides information to make that decision (see lesson #205 "Don't sign-off to approve the release of a product"). Test shouldn't say "don't ship because we can't handle concurrent users". Test should state the risks as they see them; "Shipping now would result in customers being unable to do concurrent operations". The business leaders based on information from many sources choose if the benefits of shipping outweigh the risks. Note: professional ethics encourages you to make sure customers understand what they are (or are not) getting [Hall 2009 and Berenbach 2009].

2.5 Under-testing Sequence and State

Many new testers tend to treat what they are testing in a very isolated way. While this may make it easier to think about, it frequently obscures many possible failures. Lesson #25, "All testing is based on models" is a key idea for testers. Most well-engineered systems use information hiding (http://en.wikipedia.org/wiki/Information_hiding) to encapsulate state. Testers need to have an intuition (or perhaps direct knowledge from reading the code) of the hidden state. This will reveal many more tests and possible product failures.

The incident that brought this home to me was testing the new Enterprise Java Beans (EJB, http://java.sun.com/products/ejb/docs.html). My testers had already done a good job of verifying correct sequences worked and that errors were correctly reported. But that is not enough. In most object oriented system with Exception mechanisms, an exception can ripple through the code ("writing the code that lies in between the" throw and the catch – Griffiths 2000) causing havoc ("no well-defined techniques exist for building robust exception handling into a system." – Shelton 1999). We got a bug report back indicating exceptions didn't work. It was due to the fact that a previous exception had corrupted state.

The issue with our EJB testing was we hadn't tested for sequences with errors. Does the system continue to work ***after*** returning an error? For details, see Stobie 2000, but the answer is that a short sequence can cover many requirements. The 6 requirements considered are:

1. first call - normal return
2. first call - exception raised
3. previous call raised exception - current call raises exception
4. previous call raised exception --current call returns normally
5. previous call returns normally--current call raises exception
6. previous call returns normally--current call returns normally

The first call is considered special as it goes from a never-called state to an initial state. The sequence below covers 5 of the requirements with just 7 calls.

1. Normal return from Call (Requirement 1)
2. Exception from Call (Requirement 5)
3. Normal return from Call (Requirement 4)
4. Exception from Call (Requirement 5 again)
5. Exception from Call (Requirement 3)
6. Normal return from Call (Requirement 4 again)
7. Normal return from Call (Requirement 6)

Requirement 2 and Requirement 1, by definition, can't be covered in the same test case. Requirement 6 is actually covered in most other test cases and thus not really needed here.

2.6 Skipping Coverage of the Hard Stuff

In a system logging project I worked on, I was able to cover every line of code for the logging system I was testing except under the unusual condition when the log was on a remote system using a deprecated file system. I convinced my management that this unlikely occurrence was not worth the set up costs to increase the coverage. They agreed. The only bug reported? Using a remote deprecated file system didn't work!

I misjudged the risk and convinced my management of little risk. What I would do differently in the future is consider alternative means of verification. If I can't economically set up and execute the code, then the code deserves a more careful and detailed code review and inspection (which would have caught this particular bug). Alternatively, today you should consider making sure you can mock the system being called so the code is at least thoroughly unit tested.

2.7 Over and under-testing combinations

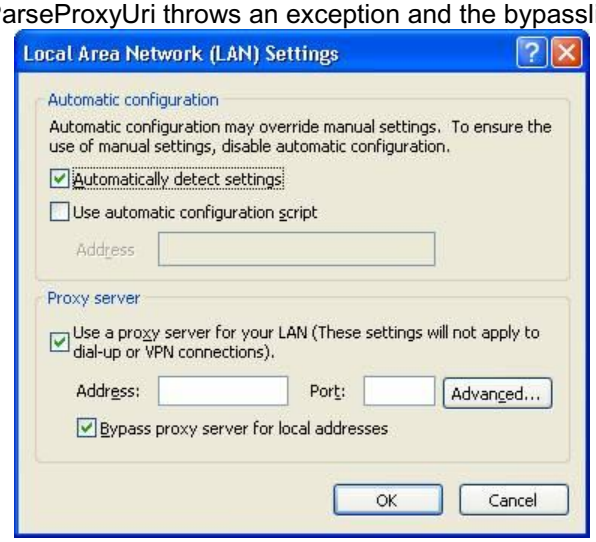
Many testers waver between testing too little by only verifying each value independently and perhaps over-testing by verifying all combinations of values even with little risk of interactions or failures. If testing all combinations is inexpensive (for example looping through a single API call that takes 1 microsecond to execute with various values) then testing all combinations is probably cheaper than even doing the analysis for less testing. However, when the combinations represent the setup or configuration of a system which may involve long periods of time to install the correct software and get it to the correct state, then combinatorics comes into play.

Empirical studies (along with a lot of war stories) show that assuming no interaction misses a number of defects. However, it also appears that given the number of additional tests needed to cover all combinations, the consequent reduction in risk isn't typically worth the cost. That is, you would be better

spending your test budget doing other things than verifying all combinations. If you know of interactions, you should test for them (see 2.5 Under-testing sequence and state above). It is only when testing for unknown interactions that combinatorics most applies. Doing pairwise testing (http://www.pairwise.org/), the all pairs approach, or using a 2-covering array (all synonyms) can help. It appears to be the best compromise between number of test cases and likelihood of revealing unexpected failure causing interactions. You can use 3-covering or high order arrays, but the number of tests grows very quickly while the likelihood of finding new failures appears to also drop quickly.

In teaching pairwise testing, I've helped other testers quickly determine over and under-testing. In one case a complicated distributed transaction system (COM+) had many configurations depending on the versions of the components. They had 80 configurations they tested and were still getting bug reports. After learning pairwise testing and applying to their array of configurations, they discovered only 40 configurations were called for by pairwise testing and the recently reported bug would have been uncovered if they had used this reduced set of 40 configurations.

Another case was the testing of a simple API that also had a user interface component. After learning pairwise testing and applying to their API for configuring a LAN, they found a bug that had existed for the past three shipped releases. Each input could take multiple values (checked/unchecked, strings, and Uris). One object, under a particular combination, did not reflect the fact that its check box was checked. If the proxy Uri is null, the ParseProxyUri throws an exception and the bypasslist is not processed.



While testing another application (Attrib.exe), another Windows team found that 13 test cases generated via pairwise analysis achieved block coverage identical to 370 exhaustive test cases.

2.8 Summary

Lessons Learned in Software Testing [Kaner 2001] presents a *Five-fold Testing System* Classification system for testing techniques with overlapping technique categories:

Who: Developers, testers, internal users, beta users, etc.
What: Coverage – Requirements, scenarios, functions, code, errors, etc.
Why: Potential problems – Risks you are testing for.
How: Activities – Regression, exploration, installation, etc.
Evaluation: How to tell whether the test passed or failed.

The key to verifying stress tests out of band lies in carefully choosing your evaluation technique, or oracle. Testing in the presence of failures covers not only evaluation techniques, but also how much coverage for each test case. Understanding the goals of your release covers all of the classifications – who will use your release (beta customers), what coverage do they need (good, clean APIs), what potential problems must be uncovered (API failures preventing usage of other APIs), and how you

evaluate if your product is ready for release. Under-testing sequence and state relates to what (coverage of the sequence and state) and why (potential problems in state corruption). Understand your coverage (what) of interactions (why) and how to most effectively test them. Finally, it is mostly about how you achieve coverage – code review, unit testing, system testing, customer testing, etc.

3 Conclusion

You can test too much in one test case causing unnecessary blocking due to bugs. You can test too much for a release by expecting more than its requirements. You can verify too much during stress testing resulting in less stress testing or overly expensive stress testing.

Testing is a business activity with a cost. Using your resources wisely to most effectively cover the risks requires you to:

- understand the product and release goals,
- deal with common issues like known errors,
- anticipate likely errors
- choose the right evaluation methods

References

Andrade, J., et al 1996 *The TUXEDO System: Software for Constructing and Managing Distributed Business Applications*, Addison-Wesley Professional, 1996

Chen, T. et al, 2004, Case *Studies on the Selection of Useful Relations in Metamorphic Testing*, HKU CS Tech Report TR-2004-13, http://www.csis.hku.hk/research/techreps/document/TR-2004-13.pdf

Berenbach, B. and M. Broy 2009, *Professional and Ethical Dilemmas in Software Engineering*, IEEE Computer, January 2009

Griffiths, A. 2000, *Here be Dragons*, http://www.octopull.demon.co.uk/c++/dragons/

Hall, D., 2009, *The Ethical Software Engineer*, IEEE Software July 2009

Hoffman, D., 1998, *A Taxonomy for Test Oracles*, Quality Week 1998
http://www.softwarequalitymethods.com/Slides/Orcl_Tax_Slides.PDF
http://www.softwarequalitymethods.com/Papers/OracleTax.pdf

Hoffman, D. 2006, *Using Oracles in Testing and Test Automation*, LogiGear newsletter.
http://www.logigear.com/newsletter/using_oracles_in_testing_and_test_automation_part-1.asp

Hu, P. et al 2006, *An empirical comparison between direct and indirect test result checking approaches*, Proceedings of the 3rd international workshop on Software quality assurance Pages: 6 – 13, ISBN:1-59593-584-3

Kaner, C., J. Bach and B. Pettichord. 2001, *Lessons Learned in Software Testing*, Wiley

Kaner, C. 2003, *What Is a Good Test Case?*, STAR East, May 2003
http://www.kaner.com/pdfs/GoodTest.pdf

Marick, B, J. Bach, and C. Kaner 2000, *Manager's Guide to Evaluating Test Suites,* Quality Week 2000
http://www.exampler.com/testing-com/writings/evaluating-test-suites-paper.pdf

Marick, B. 2000, *Testing For Programmers*, (pages 68-72)
http://www.exampler.com/testing-com/writings/half-day-programmer.pdf

Shelton, C. 1999, Exception Handling, 18-849b Dependable Embedded Systems, Spring 1999
http://www.ece.cmu.edu/~koopman/des_s99/exceptions/

Stobie, K. 2000, *Testing for Exceptions*, Software Testing & Quality Engineering July 2000.
http://www.testingcraft.com/stobie-exceptions.pdf

Software Quality Assurance in the Physical World

Kingsum Chow[1], Ida Chow[2], Vicki Niu[2], Ethan Takla[2] and Danny Brillhart[2]

[1]Kingsum.chow@intel.com, Software and Services Group, Intel Corporation

[2] Lincoln High School, Portland, Oregon.

Abstract

This paper explores the challenges of testing software on a robot in the physical world. Through the case study of running a LEGO robot with motors and sensors on a FIRST LEGO League competition table, it characterizes the environmental factors, some of which are not controllable. Through the sensors that are available on a robot, it characterizes the uncertainties from the sensor readings and the location of the robot. It then describes how it employs a software testing process to test if the robot can perform missions reliably given the less than perfect environment and sensors.

The contributions of this paper are:

1. Characterization of environmental factors in a complex system such as the physical world.
2. Establishing a method to develop the relationship with imprecise sensor readings and the functionality of the software on the robot.
3. Applying the relationship to improve the software testing process to assure the quality and reliability of the robot.

Biography

Kingsum Chow is a principal engineer from the Intel Software and Services Group (SSG). 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 he was issued 10 patents. For the last 10 years, he has been working on characterizing enterprise Java middleware and application performance and scalability on servers. Outside of work, he has been coaching a FIRST LEGO League (FLL) team since 2006.

Ida Chow, Vicki Niu, Ethan Takla and Danny Brillhart stayed together as the FLL team Nanites from 2008-2010 as middle school students from ACCESS Academy and West Sylvan Middle School of Portland, Oregon. They have recently joined Lincoln High School as 9th graders.

Published at the Pacific Northwest Software Quality Conference 2010.

1 Introduction

Testing in the physical world presents unique challenges. In the physical world, the testing environment is constantly changing which makes for differences in the actions that the system executes. When the environment changes, one must make sure that the desirable outcome is reached in all possible situations. One way to achieve this is to change the testing environment to several extremes to assure that the system performs the same in different environments so that the test plan encompasses all possible scenarios. However, this is not the only way to achieve reliability when testing systems in the physical world. Other methods include testing the system without changing the environment, relying on the unpredictability of nature to provide varied scenarios in which to test. In fact, through the FIRST LEGO League, middle school students are participating in software quality assurance in the physical world.

Dean Kamen founded an organization [1] called “For Inspiration and Recognition of Science and Technology” (FIRST). In his words, FIRST’s mission is "To transform our culture by creating a world where science and technology are celebrated and where young people dream of becoming science and technology leaders." The leaders of FIRST believe that in our world engineers and scientists currently have the ability to create positive change and revolutions that may alter the way our world operates. The key objectives of FIRST are to get young people involved in the world of engineering and to inspire young people to realize the importance of engineering and technology.

The FIRST program consists of a community that drives for creativity and success among kids all over the world. Within this program is the FIRST LEGO League (FLL), designed to push for innovative solutions and ideas that stand out. FLL is a competition that judges what a team has accomplished by how unique it is and the impact to the community. This way of judging makes kids strive for perfection and imagination, creating ideas that might have never been thought of by adults. With different themes ranging from nanotechnology to green environments each year, it gives a new topic for children to learn about and share with others. The table missions each year are a variety of different tasks that involve triggering machines, delivering and retrieving items, or a combination of both. When teams run missions on the table, there are stationary objects on the table, some of which are secured and are easier to navigate around, and some loosely placed objects, which if are accidentally moved could cause navigational problems. Additionally, there are also different factors on the table, such as the table surface and lighting conditions that may affect the way the robot performs. Some of the problems that we have encountered on the table include; varying table lengths and widths, changes in light intensity and the direction in which light comes from, different types of wall and mat surfaces, and diverse battery levels.

FLL robots and table missions are an effective model for software testing. In FLL we use our LEGO Mindstorms [2] technology to build and program a robot. Using math and science, along with critical thinking, we are able to create and utilize innovative solutions to solve difficult problems. When programming the robot [3] we must ensure that the robot can reliably navigate around the table and perform missions regardless of what unexpected changes it faces.

2 Methodology

Software quality in the physical world is not just the software, but also the interaction among the software, the hardware and the environment, i.e., Software Quality in the Physical World = Software x Hardware x Environment.

In FLL, a team can fix the hardware design using a set of LEGO pieces, motors and sensors. But the quality and performance of the robot still depends on the environment [4] that is not necessarily under their control. These environment factors include differences in how the table is constructed and the ambient light condition. As some environment condition is not under control, there is variance from run to run even using the same robot and the same software on the same table. To reduce the variance and increase the confidence of a working robot, more runs need to be made on the same table. To increase the assurance for it on other tables, most tables need to be used. The combination of test cases, multiplied by environment variables, quickly gets out of hand. The ultimate question is how to get the most out of testing without making too many runs.

2.1 Environmental factors

The environment for the robot can be divided into static and dynamic categories. The static factor is the one that doesn't change within the test environment. For examples, the light sources, the table, the mat and the fixed structures on the mat can be treated as a constant given a test environment. The dynamic factor is the one that changes within a test environment. For examples, the stray light source and the location of loose objects on the mat are treated as variables.

2.2 Imprecise sensor readings and the functionality

In planes, there is something known as the flight data recorder. During the duration of the entire flight, this data recorder [5] constantly collects the data values from all of the equipment on the plane. In the case of a malfunction aboard the plane, the data can be used to resolve the problem of what caused the malfunction. This is the method we employ to develop the relationship of imprecise sensor readings and the functionality of the software. The flight data recorder is also known as black box. Black box is looking at how a program might malfunction while not being able to examine the source code for the program. Thus we call our method "black box data analysis". This method can be used to test for the dynamic and static factors that may affect the program, Black box will take data from all the sensors on the robot throughout the running of the program. Reviewing this data can show what varies in different runs, and what factors cause a malfunction.

2.3 Software quality assurance in the physical world

In order to apply this to the robot, there is the problem of limited memory in the NXT brick, and to counter this, we cannot collect data points constantly, as is the case in airplanes. As a workaround, the collections of data points only happen at certain checkpoints. Checkpoints are points in the program that are of importance to the success of the program.

3 Case Study

In FLL, a team of students are given a table containing a number of physical tasks they must complete. Some of these tasks require the robot to pick up a loop or knock down an obstacle. To do so, the robot must be able to navigate reliably around the table and that is the main focus of this paper. At this point, one can assume that their robot is not fully reliable and that there are a large number of factors causing the unreliability. For example, it may not have a sturdy chassis or uncontrollable environmental factors such as stray light sources could cause the robot's sensor readings to be off. In the interest of reliability, it was found that implementing a black box into the code helped improve the performance in the long run. During our tests, the robot recorded sensor values at pre-defined check points on the table. The checkpoints are generally landmarks, so we know where the data is taken. After the robot achieves a perfect run, we archive the data to compare to when imperfect runs appear. A perfect run is defined as a run where the robot accomplished everything it was programmed to do without error. Once the archive of imperfect and perfect runs has a sufficient amount of data, e.g., more than ten runs, ranges of perfect run data can be created. We can derive other useful statistics such as standard deviation and 95% confidence intervals. Knowing the confidence intervals for the sensors at a given check point, it becomes possible to program the robot to alert its operator, or even correct itself, when sensor values are outside of their normal ranges. This is done purely through an additional piece of code that checks all sensor values at each checkpoint.

In the FLL case study here, we first describe our experience in testing line following programs and then follow by testing the robot for the entire challenge, combining line following and other missions.

3.1 A simple test case – testing line following programs

To illustrate the problem of testing a program on a robot, we start with a simple program that involves the environment in several ways. The program follows a black line on the mat and tries to follow the line as close as possible and within a reasonable amount of time. Line following is one of the techniques used for robot navigation. An adequate way to test line following is indeed important.

The line following program depends on several environmental factors: the ambient light source, the changes in lighting on the mat, the actual colors and the uniformity of the colors of black and white on the table. The texture of the mat and the flatness of the mat also play a role in affecting the line following activity. While in FLL an icon-based programming language called NXT-G is used, for the sake of simplicity, we converted the NXT-G programs to RobotC to illustrate this case study. RobotC is not used in FLL competitions, but RobotC is really similar to C and we believe most readers of this paper would find it easier to understand RobotC programs.

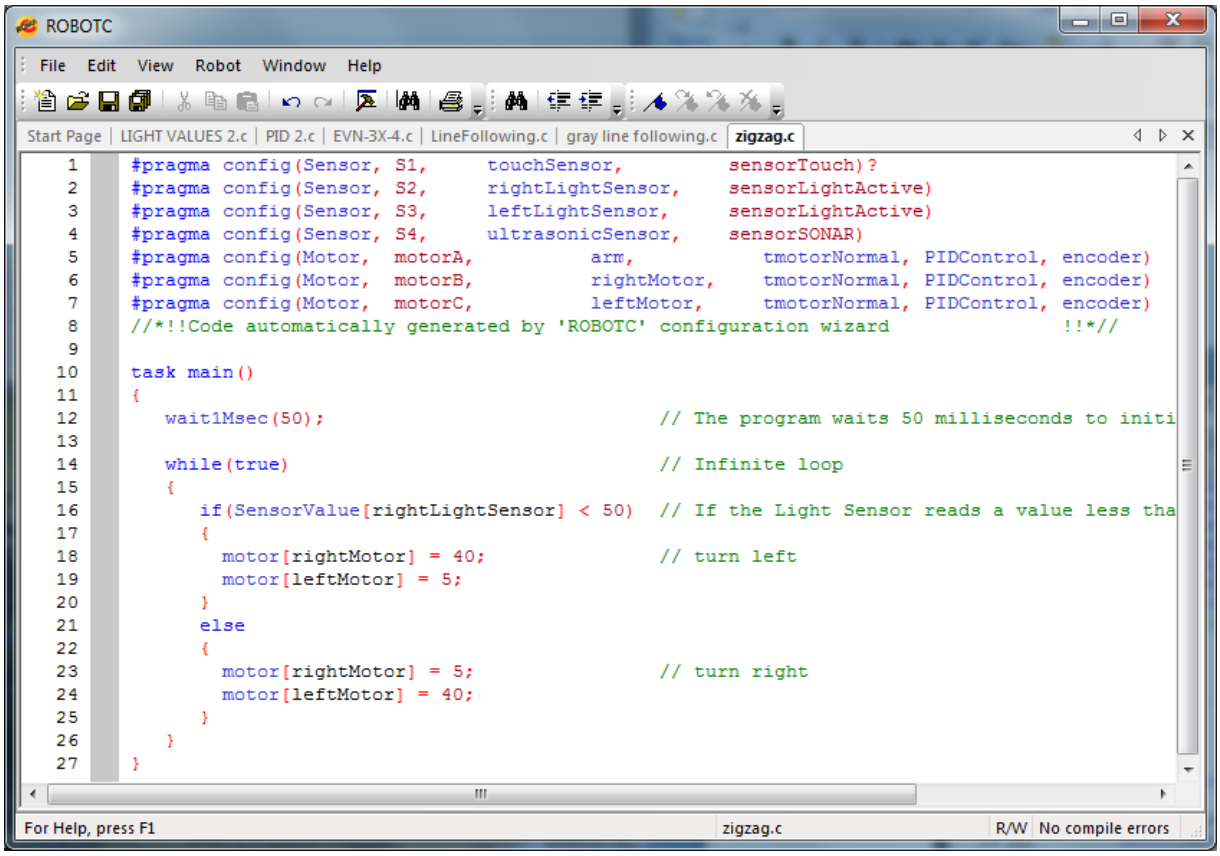


Figure 1. A zigzag line following program

A zigzag line following program (Figure 1) illustrates one of the simplest line following programs written in RobotC and testing even that is not simple. The program defines the four sensors attached in lines 1 to 4 and the 3 motors from line 5 to 7. Without going into the specifics of sensors or motors, it is suffice to say that the sensors generate readings for the robot to act on while the motors control the action the robot is to perform. Note that NXT servo motors also come with rotational sensor readings, i.e., there are actually a total of 7 sensors, 4 are standard alone sensors and 3 come with the motors. The program will check if the light sensor on the right sees something that is too dark (< 50) then it will turn left, or something that is too bright, and then it will turn right. 50 is a reasonable midpoint between bright and dark. The motor speeds (40, and 5) for the turns are arbitrary. In order to test such a program, we would run the program multiple times in many different cases including different starting robot positions to make sure that the program will run under a variety of conditions. However, even if we write down the starting positions of the robot we would like to test, there is still small variation in the way we put the robot down and also almost infinite different kinds of black lines that we need to test with. Yes indeed, there are already too many scenarios to test even for such a simple program.

While the zigzag line following program is sufficient for some simple tasks, it is spending a significant amount of time turning, rather than making progress in going forward. A well known gray line following program solves the deficiency of zigzag line following by introducing a gray region, and within that region, the robot should simply go forward, i.e., turning on both motors at the same speed. Figure 2 illustrates such a program.

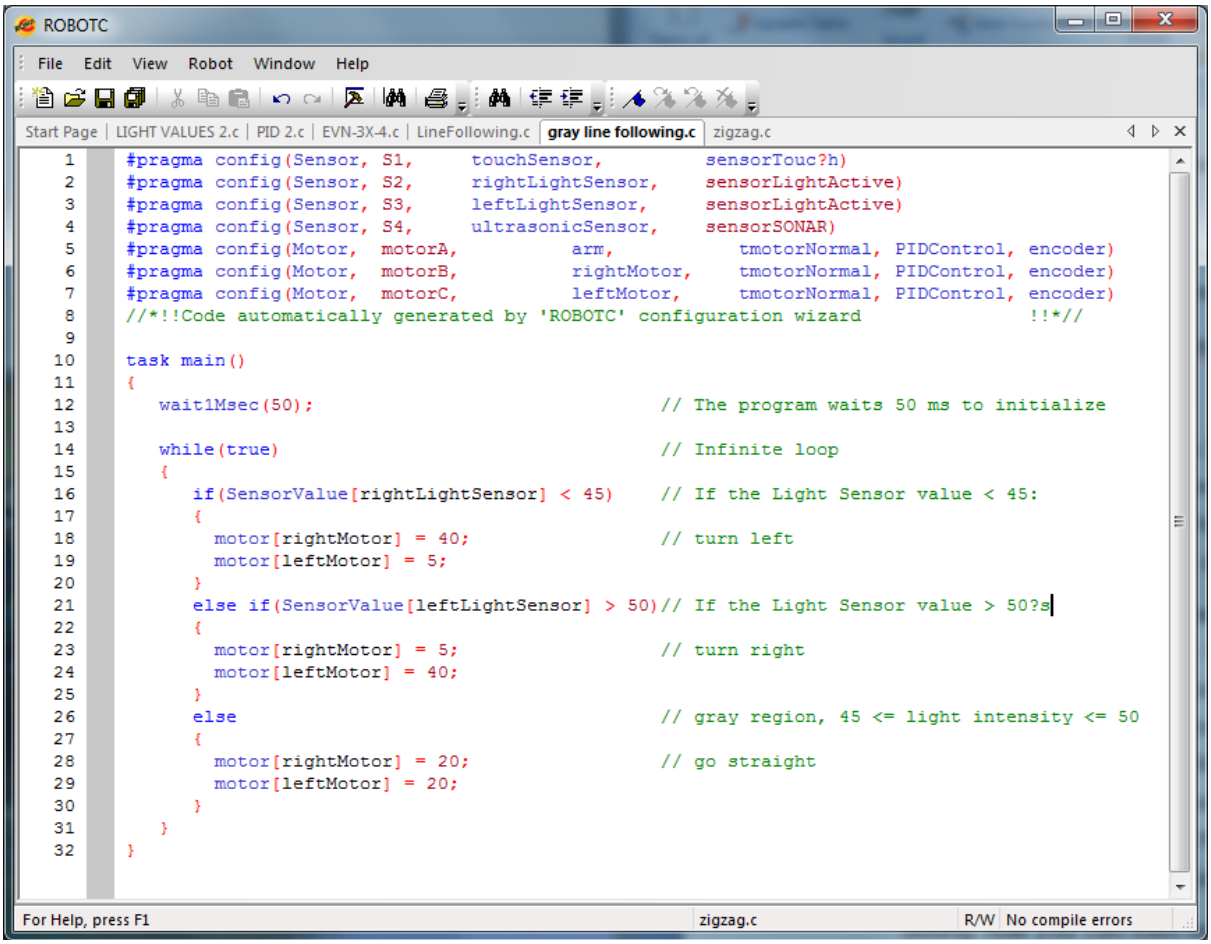


Figure 2. A gray line following program

The main difference between the two programs is captured in the else part, i.e., lines 26-30. The gray line following program will simply move forward within the light sensor reading range of 45 and 50. Given the two line following programs, how do we test which program is better for the robot? Intuitively most people would agree time and accuracy would be the two most important criteria. We can easily time the robot from a starting point to an end point. But how do we measure the accuracy of following the line? It is not uncommon for some teams to decide to just observe the run and if the robot is seen following the line by several people, then we call it following the line. Our team took a different approach. We decided not to rely on human judgment at the table but instead we log the sensor readings throughout the entire run while following the line. We checked if the sensor readings are close to the midpoint of the line we are following and add up the sum of the differences between the observed sensor readings and the midpoint. In a way, if the robot is moving fast, fewer data points would be collected and the sum of errors may be small even if individual errors might be big. We believe this is a good way to measure accurate and fast results. We are still exploring other better ways.

One short coming of the gray line following program is the need to slow down the forward move so it can detect an error, e.g. making a sharp turn, before it is too late. To overcome that problem, we implemented PID based line following program that greatly improves the accuracy and performance of line following. Still, line following is just a small component of testing robots in FLL. We are going to describe how we test the entire challenge mission next.

3.2 A complex test case – testing all programs for the table competition

For the LEGO Mindstorms NXT robots used in the FIRST LEGO League tournaments all across the world, a plethora of problems are encountered throughout testing the programs. Different ways of testing are employed in order to find as many of the flaws that are existent in any robot.

In testing the multiple missions in FLL, we first map out the programs that will be executed. What the programs do, the order they are executed, and how they are implemented are different for all the FLL teams. However, all the teams face the same challenge of testing their programs on their own table and the robot is still expected to perform in the tournament at a table that is the same according to specification but different because of the static and dynamic factors described earlier.

We attempted to test the static environment variables by constructing models to simulate environments that are not exactly the same as our own table. We developed four test environments based on our experience how a robot that works on a table may fail at a different (but built according to specification) table.

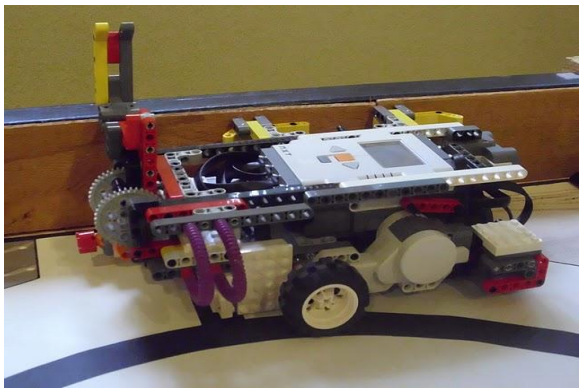


Figure 3. Rough Wall Simulation

Figure 5. Shorter Wall Simulation

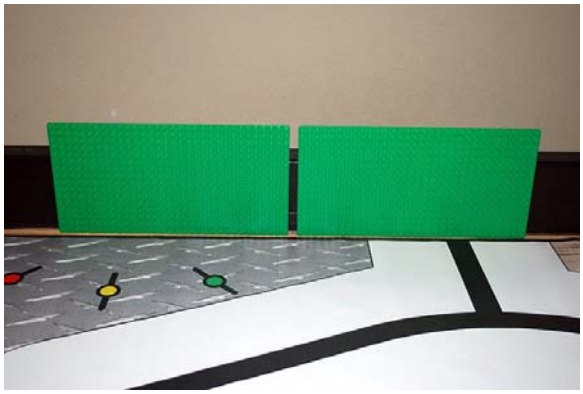


Figure 4. Wall Crack Simulation

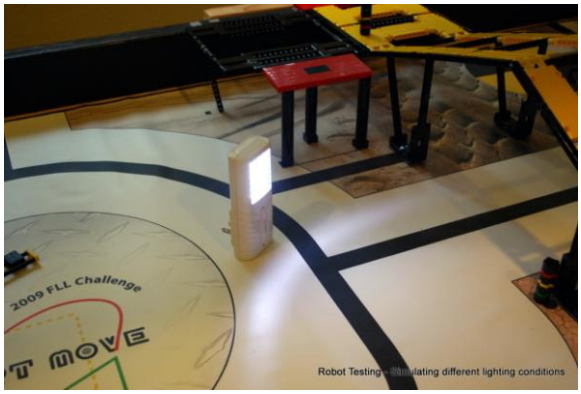


Figure 6. Extra Ambient Light Simulation

Figure 3 illustrates changing the environment to test the limits of a robot includes the use of a rough extra wooden board added to already existent wall, in order to see the effects of this surface on the movement of the robot. The rough board also tests how the robot is able to glide along the differently textured barrier. Figure 4 illustrates a test to check how the robot responds to a crack in the wall. The gap between two large green LEGO plates can simulate if a robot that is gliding on a wall

will get stuck in a crack like that. Figure 5 illustrates a test if a table is slightly shorter in length. A shorter table may interfere with sensor readings and may show the limitations of room that might be introduced at the actual tournament runs. Figure 6 illustrates a test with different lighting condition and a shadow is introduced. A bright LED is used in an attempt to drag bad light readings into the sensor as the robot follows the line. In actual tournaments, a strong light can be shining from above, as that allows the tables to be more visually pleasing for the media that is present at tournaments.

3.3 Testing static and dynamic environmental factors

After overcoming testing of a specific challenge like line following, and fixed environmental factors, we realized that the number of test cases had already increased exponentially. It was impractical for us to continue testing the robot in the same way and hope to complete the necessary tests within a couple of months, while debugging and fixing many other issues. We resorted to the method we called “black box data analysis” in section 2 to combine situations in a way we can act on.

In our method of black box data analysis, data are taken from all five sensors at specific check points in the program. They are then uploaded to the computer for analysis. Each data point is categorized into which checkpoint it was collected from, which sensor it comes from, and which run of the program it came from.

Table 1. Black Box Data Collection for NXT Example

	Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9	Run10
A	-49	-49	-35	-19	-6	1	1	-2	-2	-2
B	1424	1427	1426	1424	1425	1423	1424	1416	1417	1416
C	1408	1410	1409	1406	1406	1409	1406	1404	1405	1404
2	41	40	41	41	40	40	40	40	40	40
3	54	56	55	55	54	55	55	54	54	55
A	443	443	457	474	487	493	494	492	492	492
B	2099	2105	2101	2099	2097	2099	2100	2103	2104	2100
C	2167	2173	2168	2169	2163	2172	2168	2177	2174	2168
2	25	24	24	24	24	24	24	25	24	25
3	36	36	36	36	36	38	39	38	36	37
A	476	476	490	436	469	461	462	460	459	461
B	3321	3328	3322	3324	3320	3322	3326	3328	3328	3323
C	3376	3383	3378	3379	3371	3379	3380	3391	3386	3380
2	25	25	25	25	25	25	25	25	25	25
3	62	62	62	62	62	62	62	62	62	62
A	-6	-5	10	-29	-3	-4	-4	-6	-6	-6
B	4857	4867	4865	4857	4891	4867	4867	4870	4870	4867
C	4976	4986	4988	4978	5007	4950	4947	4956	4951	4948
2	26	26	32	26	26	41	35	35	39	39
3	61	61	61	61	58	61	61	61	62	62
A	296	298	312	268	298	292	293	290	291	291
B	3556	3566	3543	3557	3782	3587	3580	3584	3591	3589
C	3558	3567	3552	3558	3778	3538	3533	3541	3538	3536
2	39	40	38	39	25	39	38	39	38	38
3	62	62	62	62	37	62	62	62	62	62
A	-34	-32	-19	-33	-13	-12	-11	-14	-13	-13
B	3138	3147	3125	3135	3358	3167	3160	3164	3170	3168

C	3140	3148	3135	3140	3359	3121	3116	3122	3120	3117
2	24	25	25	25	48	25	26	25	25	25
3	62	62	62	62	46	62	63	63	63	62
A	0	0	16	-21	4	0	0	0	0	0
B	2439	2446	2423	2437	2662	2465	2459	2464	2474	2474
C	2436	2444	2432	2438	2658	2416	2415	2421	2419	2416
2	50	50	50	50	50	50	50	50	50	50
3	63	63	63	63	64	63	63	64	63	64
Blue	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Grey	Y	Y	Y	N	N	Y	N	Y	Y	Y
Red	Y	Y	Y	Y	N	Y	Y	Y	Y	Y
Brown	N	N	N	Y	Y	Y	Y	Y	Y	Y
Beacons	8	8	8	8	8	8	8	8	8	8
Base	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
Battery	7.9	7.9	7.9	7.9	7.8	7.8	7.8	7.7	7.7	7.7

Table 1 shows how the data is collected. A, B, C, 2, and 3 all represent different types of sensors. A, B and C are rotational sensor readings associated with motors A, B and C. Sensor readings 2 and 3 are left and right sensor readings. Each group of the horizontal rows of “A, B, C, 2 and 3” is a set of data read at a given check point. In the table, 7 check points are taken in a given run. The columns contain different runs of the data. A total of 10 runs were made. Near the bottom of the table, the rows “blue, grey, red, brown, beacons, base” all describe the success conditions for the FLL challenge. A “Y” means that component of the challenge is completed and an “N” means the robot fails to achieve that component of the objectives. The battery section of the table lists the voltage level at the time of the run. We used the data from the successful runs to determine the range of working sensor values for the check points. We then checked if the runs that failed had values outside the confidence intervals of the successful runs. We also implemented a warning sound in the program should the robot predict that it is going to fail based on the confidence intervals. We believe we have only scratched the usefulness of the “black box data analysis” approach and ~~we~~ plan to refine it in our next experiments.

4 Discussion

Robot Software [5] consists of the instructions that control a robot's actions and provide information regarding required tasks. When a program is written using this software, the robot is able to execute commands and perform tasks. Programming robots can be a complex and challenging process, and while it has become easier over the years, the lack of cross-platform industry standards has affected the development of software tools for robots compared to other automated control systems.

Software quality assurance in some way is inspecting the products in order to determine whether they meet the requirements and also detecting the defects of the system. In reality, there is no perfect test system that can detect all defects. It just reduces defect risks as much as possible. The ISO 9126-1 software quality model [7] identifies 6 main quality characteristics, namely functionality, reliability, usability, efficiency, maintainability, and portability. In this paper, we focus on functionality,

reliability and efficiency. Hwang [8] describes an alternative method to adapt software quality assurance to robotics.

Nakajima et al [9] describes a software design contest involving the design of software to automatically control a line-trace robot, and conduct running performance tests was held. They found that the quantitative measurement of the structural complexity of the design models bears a strong relationship to qualitative evaluation of the design conducted by judges. However, there is no strong correlation between design model quality evaluated and the final system performance as the software judging might not have taken into account of the algorithm used to handle environmental variables. Our method here does include a testing process that would enable the correlation of the black box data analysis and the outcome of the robot performance.

5 Conclusion

This paper describes the authors' experience in testing software quality in the physical world using a case study based on FIRST LEGO League competition. After staying together five years as a team, we have been exploring ways to improve the testing process to find out if the robot is going to perform on a different table in the competition environment. We worked on the characterization of environmental factors in a complex system such as the physical world and establishing a method to develop the relationship with imprecise sensor readings and the functionality of the software on the robot. We applied the relationship to improve the software testing process to assure its quality and reliability.

Acknowledgments

Moss Drake helped review and improve the quality of this paper.

References

1. FIRST. http://usfirst.org/
2. The LEGO Group, LEGO Mindstorms, http://mindstorms.lego.com/
3. Joseph L. Jones, Robot Programming – A Practical Guide to Behavior-Based Robotics, McGraw Hill 2004.
4. Ida Chow, Vicki Niu, Ethan Takla and Danny Brillhart, Designing and Testing Robots for Reliable Performance, FIRST Robotics Conference, April 14, 2010, Atlanta, GA.
5. Flight data recorder. http://en.wikipedia.org/wiki/Flight_data_recorder
6. Robot Software. http://en.wikipedia.org/wiki/Robot_software
7. ISO 9126 Software Quality Model. http://www.sqa.net/iso9126.html
8. Sun-Myung Hwang, Testing Method for Intelligent Robot Software Components Testing, http://staff.aist.go.jp/t.kotoku/conf/iros2006ws/WS4-6.pdf
9. Eiji Nakajima, et al, "Experiments on quality evaluation of embedded software in Japan robot software design contest," pp.551-560, 28th International Conference on Software Engineering (ICSE'06), 2006. http://www.computer.org/portal/web/csdl/doi/10.1145/1134285.1134363

Visualizing Software Quality

PNSQC 2010 conference paper

Mark Fink

August 14, 2010

Author contact: mark@mark-fink.de

Project website: http://www.testing-software.org

Abstract

When software projects reach a considerable size (multi-person, multi-man-year) they are difficult to maintain. Which results in high efforts for software maintenance, code duplication, bugs, and other quality related problems. The consequences include limited software life cycles, especially in areas where software evolves rapidly. For example in an industry setting, business applications are usually re-implemented every 3-5 years. This is commonly considered as the only foolproof strategy currently available. It is also considered as a huge waste of resources and a true sign of the software crisis.

Quality initiatives could significantly improve the software quality, but they usually suffer from tight budgets, lack of support and other neglectful factors. Also, due to the fact that software can be immensely complex, complete testing is impossible. So it is desired to drive testing efforts to parts of the software product that would benefit the most (risk considerations).

The root cause of the software crisis lies in software engineering itself. Software is the most complicated product we make and we can not see it, or touch software like we can see or touch the work products of other engineering disciplines, such as construction or mechanics. During visual inspections you instantly see whether a car misses the bumper, a door or the wind-shield. The quality of a software project is not as obviously apparent through visual inspections.

What if we could make the quality of software visible? This could be used to govern testing initiatives and could help to decide if the software is really ready for release. This paper provides in-depth insights and experience on interactively visualizing different aspects of software quality.

1 Introduction

I started the visualizer tool project in fall 2009 in order to create interactive visualizations of software quality for huge code bases, like Mozilla Firefox, Apache Webserver, and the Python code base. When you are working on huge code bases like these and you need to improve the quality, it is difficult to determine where to start. Most software quality initiatives have limited budgets these days. As a consequence you want to identify which parts of your code base would benefit the most from your initiative. It is also desirable to interactively visualize how the quality of your project evolved over time.

The times that one single developer could know the complete code of the whole system are long gone. For example the first Cisco router operating system had 30.000 lines of code and was written by one developer. Today's router operating system size is somewhere between 50 and 100 million lines of code. This is over 2.000 times as much code for maintenance, testing, etc. Today a whole team maintains the router OS. One of the reasons for this increase in size is that developers in the early days usually had to shrink their code to fit onto the hardware. The limited hardware resources gave them a strong incentive to clean up their code during maintenance.

When testing huge software systems an enormous amount of data today is generated, especially with test automation and continuous integration. Analysis of this data is a huge task. New methods are needed to analyze this data and to extract useful information from that data.

I think analysis of data is one of the key elements/ skills in software quality assurance. Since software systems dramatically increased in code size they usually require team work for development, testing and maintenance. As a consequence none of the team members has an overview about what needs testing/ retesting any more. The need for analysis tools arises that support the decision making process on what needs testing, and this is where visualization can add value.

The science of Data Visualization itself is an emerging discipline. Mankind has a long but erratic history with data visualization, shaped by the available technology and by the pressing needs of the time. Its modern roots date back for example in fighting the Cholera Epidemic in London 1854 [1]. The last decade have brought new and useful innovations in visualization techniques. Many innovations in this area were discovered in Genetics.

This paper gives an insight into the application of the discipline of software quality visualization. The focus is on discussing and documenting the requirements for an interactive software quality visualization tool with the aim to improve the software development process. A prototype of the tool has been implemented in order to demonstrate the interactive visualization of software quality. Some organizations currently are thinking about adapting software quality visualization, some have already managed to successfully apply them [2] and I give examples where possible.

2 The problem

The software crisis has roots in the nature of software itself [3]. Software-development is extremely powerful, so we can create complex application like Internet Browsers, Word Processors and massive simulations. But programming is also immensely difficult because software is the most complex product we make.

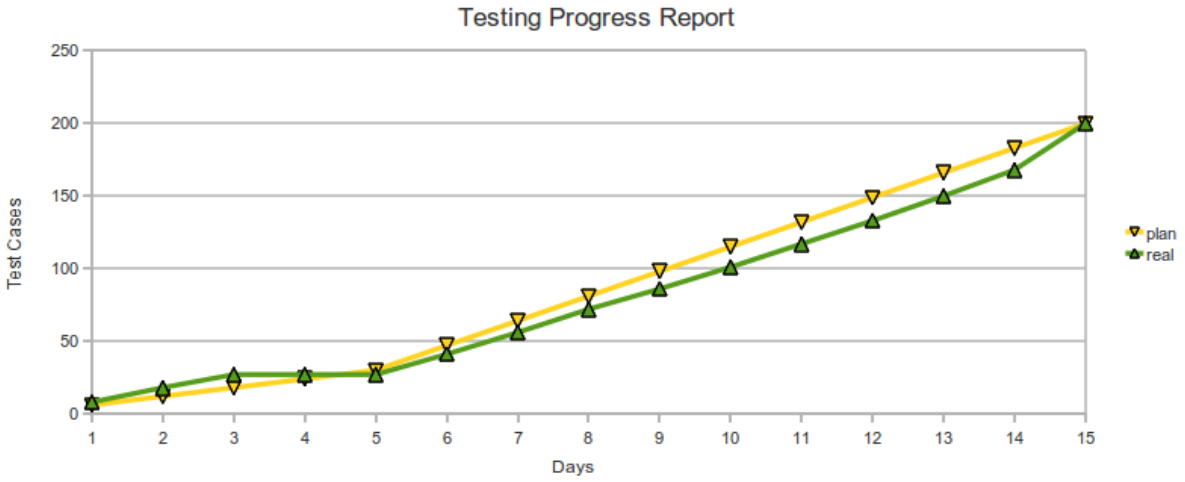
The management of a software project is very difficult, too. We still cannot agree on appropriate metrics to measure the quality or completeness of a software project. Here I believe that lines-of-code is absolutely inadequate for various reasons. First of all, effort estimation is not possible; it is impossible to determine at the beginning how many lines of code the system will have. Second, it is extremely difficult to say how much time it will take to get the lines done.

"Measuring programming progress by lines of code is like measuring aircraft building progress by weight", Bill Gates.

When we look into testing as part of software construction, we already know that we can never fully test a software package of significant size because of limitations of resources like time and budget. Therefore we urge to put our sparse resources to effective use in order to maximize the benefits. We want to focus our testing resources on the parts of the software where we have the most risk.

In industry software projects we usually get a very different view. Let me explain. During my career in software testing I have been facing an ongoing phenomena which I call "The illusion

of testing resources well applied". A good example is during the maintenance phase of a software application: the test analysts come up with a sound test plan based on the design of the software change. The test plan contains test cases for the new features, plus regression test cases and an effort estimate for testing (in the sample 200TC, 3 testers allocated for a 3 weeks time frame). After the testing resources are allocated to the project the test manager assigns the test cases to these resources. Test progress is usually reported to management in the following form of a progress chart:



This almost scientific appearance of the process with the above diagram, which is used in management reporting, gives the impression that everything is in order. But on the contrary, testing resources are not applied efficiently! The fundamental flaw of this approach is that it does not contain any information about whether the test cases make sense, if they cover the software changes, or risk considerations. Unfortunately the review process for the test plan often does not reveal these problems. In practice the review process is often used to shift the responsibility away from the (outsourced) testing department.

This is really a planning/ analysis problem. In order to significantly improve the testing process and as a consequence the testers contribution to the project we have to apply better analysis and planning techniques.

A planning/ analysis tool like the visualizer should help you identify:

- Areas in the codebase that have been changed
- Areas in the codebase that require intensive testing (identify risks)
- Areas with low unit test coverage for test automation
- Performance critical spots in the codebase

3 Software Quality Visualization

Software Visualization is a topic that has been discussed in computer scientist cycles for many years. The broader topic of information visualization received a lot of attention e.g. through the breakthrough in the genome project made possible by visualization. The sheer amount of data we collect every day makes visual analysis skills a key competence for today's companies.

Some industries adapted visualization techniques a long time ago. For example air-traffic control could hardly be imagined without a graphical display. And Wired Magazine has long run a feature called "Info Porn", which attempts to showcase data visualization in interesting and unusual ways.

Discovery and analysis of data through visual exploration was made popular by Prof. Edward R. Tufte's books. The principle is to map the attributes of an abstract data structure to visual attributes such as Cartesian position, colour and size, and to display the mapping [4].

In the last two years some companies reported success in adapting visualization techniques to software testing [2].

3.1 Treemaps

The key to visualization principles is to understand the enormous capacity for human visual information processing. The human eye/ brain system is capable of processing thousands of information bits in a few milliseconds if presented visually. It does not take any effort to see whether there are one or several red dots and it can be done in less than 200 milliseconds [5].

The use of proximity coding, plus colour coding, size coding, animated presentations, and user-controlled selections enable users to explore large information spaces rapidly and reliably [5].

Treemaps are excellent for mapping arbitrary two dimensional data to hierarchies and for seeing relationships between the data and the tree [5]. Given primary graphical encodings of area, colour and enclosure, treemaps are best suited for the task of outlier detection, and cause-effect analysis. But without adequate computer support, they are difficult to generate by hand. Only recently have adequate support routines and libraries emerged that allow easy generation of treemaps.

Treemap implementations today appear as one of several different treemap layouts. Common Layouts are: Clustered, squarified, and slice & dice.

Treemap visualizations tend to scale up to support more items than other visualization techniques such as bar or radial graphs, in some cases they can are used to display more than one million items [5]. This is sufficient to handle the files or functions of the biggest software projects, like operating systems or applications like internet browsers.

I believe that treemaps beautifully implement the information visualization mantra coined by Ben Shneiderman: "overview first, zoom and filter, then details on demand" [5].

For more details on Prof. Tuftes Visualization principles and on Treemaps please refer to the excellent PNSQC 2009 paper and slides by Marlena Compton [6].

4 Firefox as a sample code basis

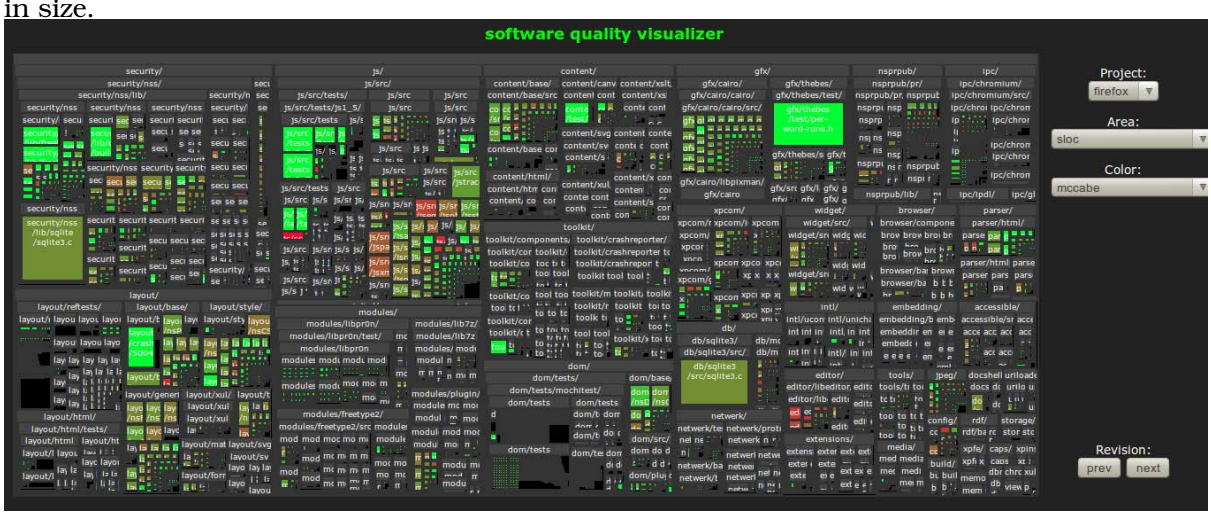
Many circumstances made the Mozilla Firefox browser the ideal reference project for the development and demonstration of the visualizer package.

- Open source project with code openly available
- Defect database openly available
- Significant size of the codebase (> 1 Mio. source files, > 50 Mio. lines of code)
- Proficiency in automation and mature quality assurance
- Technology affluence
- Existing contacts with people from Mozilla QA team; looking for feedback on the prototype

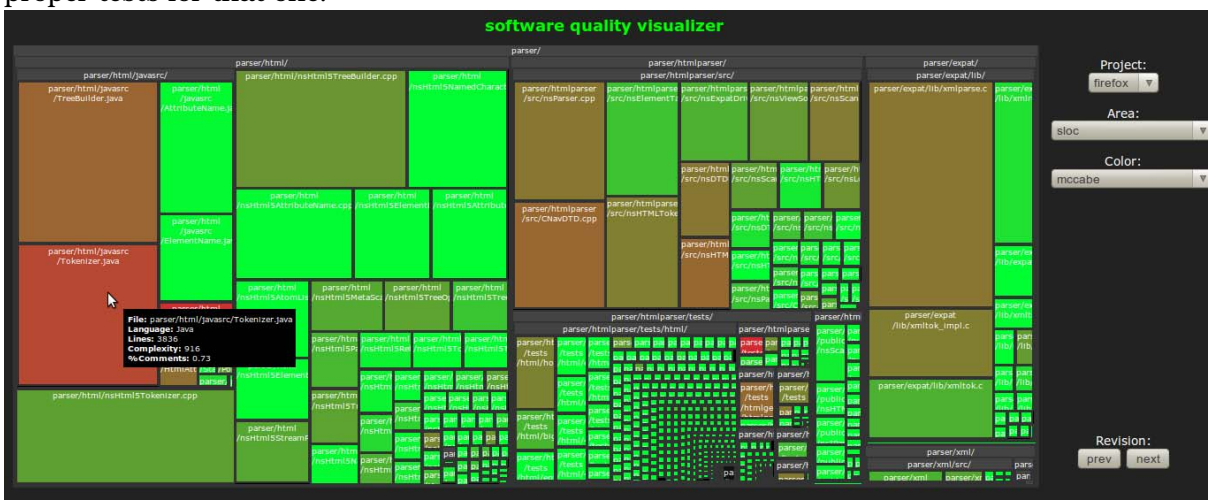
4.1 The visualizer demo

A significant part of the conference talk will be the demonstration of the features of the current visualizer implementation. The demonstration of the visualizer features will be performed along the lines of the visualization mantra (overview first, zoom, details on demand). Also the navigation of revisions will be demonstrated.

The following screen-shot shows the top level visualization of the Firefox source code. In this view we quickly familiarize about the structure of the project code and how the modules relate in size.



The next screen shows the visualizer after the user navigated one level down into the Firefox *parser* module. In this view the user can easily spot complex code like the *Tokenizer.java* which has over 3.800 lines-of-code and a McCabe complexity of over 900. Obviously the file contains a huge amount of switch statements. A comment-to-code ratio of 0.73 shows that the file maintains a sound level of documentation. But we have to make sure that we have proper tests for that one!



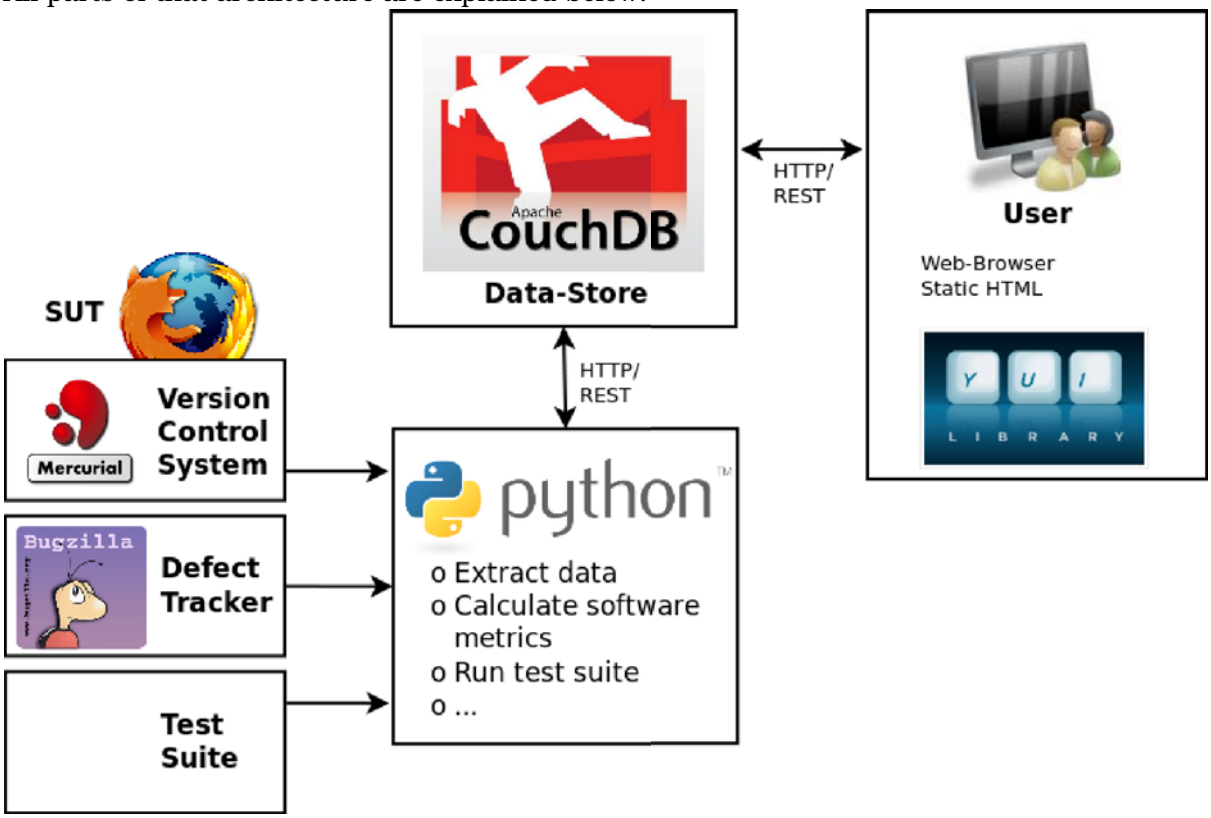
The screenshots above also show some of the features of the current visualizer implementation:

- **Select box for projects** - you can add another project to your client by providing a name and the URL of its data store which contains the metrics data
- **Select box for the metric used for area** - usually the lines-of-code metric but it is possible to visualize other metrics in relation
- **Select box for the metric used for colour** - select one of the metrics contained in your project data store. McCabe-complexity or comment-to-code ratio are good candidates to start your analysis.

- **Buttons to navigate revisions** - navigate project source code revisions in order to analyse how the project quality evolved over time
- **Context help** - provides the detailed metric information on demand

5 The visualizer architecture

The following diagram gives an overview of the architecture of visualizer tool implementation. All parts of that architecture are explained below.



5.1 System Under Test (SUT)

The SUT is the software project of which we want to analyse the software quality. Besides the source code of the software which is usually stored in a version control system, there are usually more development tools that provide important data for visualization of software quality. The basic set of tools is the version control system, the bug tracker and the test suite.

5.2 Data Collection

The metrics [8] package which is also maintained by the author is used to extract metrics from the code base and to walk the version control system (Subversion (SVN), Mercurial (HG)). Currently the metrics package provides metrics on lines-of-code, lines-of-comments, and McCabe-complexity consistently over multiple programming languages (for example C, C++, Java, Python, shell scripts, JavaScript, and many more). Existing metrics packages like SLOCcount, PyMetrics etc. could not be used because each metrics package has its own algorithm on how to count lines-of-code. Extraction of data from the defect-tracker and test-suite tools also belong to this part of the architecture but are not yet implemented.

The Data collection scripts have been implemented in the Python programming language because the available libraries make it particularly interesting as a glue language.

I assume that from your own experience it is clear to you that the data collection will be different for every project. That is also the reason why the visualizer tool will never work completely out of the box. The visualizer project aims to provide some common tools and infrastructure in order to ease the integration in your project environment. Data collection will still be highly dependant on your project environment and there will be at least some customizations necessary in order to access your project tools like version control system, defect tracker, continuous integration systems and test suite(s).

5.3 The data store

Nowadays successful companies like Google, Amazon, ebay, Yahoo, Facebook, etc. owning big applications, abandoned their RDBMS systems. Instead they are implementing their own flavour of document oriented or key-value oriented data stores. The main difference to RDBMS is that these new systems, which are commonly called NoSQL systems, scale better for big data stores. Meanwhile successful Open Source NoQSL implementations are also available.

Map/Reduce is a concept developed at University of California, Berkeley that applies functional programming concepts to the problem of big data analysis. CouchDB is an Erlang based open source implementation of the Map/Reduce concept by the Apache Software Foundation [9]. CouchDB has no Joins, no Relations, no Tables, and no SQL which are the RDBMS concepts that we all know from traditional databases. What makes CouchDB a particularly good fit for the visualizer implementation is that it uses JavaScript for writing the Map/Reduce functions and that it provides a REST API that plays nicely with the Ajax frontend of the visualizer. This means all the communication is plain HTTP. Additional middle-ware is not required.

The data store is one of the key elements of the visualizer architecture. As it turned out the CouchDB performance is not as good as it occured to me at the beginning and it is also difficult to implement the advanced visualizer features like interactive queries in CouchDB. I currently re-evaluate CouchDB if it can be adapted enough to serve the needs of the visualizer tool or if it has to be replaced altogether by another data store.

5.4 Interactive visualization (frontend)

A key concept to the visualizer is that it has to be highly interactive, following Ben Shneidermans visualization mantra “overview first, zoom and filter, details on demand”. This is why the visualizer requires an implementation that supports the user’s interaction with the visualization data.

Another factor that has constrained the technologies being used for the visualizer implementation is that all Mozilla QA tools must be web based like for example Graph-Server, Tinderbox etc. The Firefox development is Open Source and therefore it is very rarely the case that two developers are co-located, except maybe for sprints. This requires web based tools that provide easy access for the developers.

For the prototype implementation the Yahoo User interface library YUI3 was used. YUI3 is a Javascript/ Ajax framework [10] which provides a sane module concept and proved to work very well during the implementation of the prototype.

6 Quality metrics

A metric is something that you can measure, and used as a method in order to reason or infer about something you can not (easily) measure. For example lines-of-code is a metric that is often used as a proxy to predict efforts/ cost.

Why measure? In everyday life we use experience, common sense, and intuition to come up with solutions for our problems. We measure because software development is so complex that intuition alone is not enough. Team work makes it even more difficult because the single expert with an overview does not exist any more in today's projects.

"Measurement is the key to all disciplines of science and technology, and the maturity of the discipline is marked by the extent to which it is supported by a sound and comprehensive system of measures, measurement standards, measurement tools and measuring procedures" [11].

Usually the software development projects have a lot of data that would be useful for quality analysis. The problem is that it often proves difficult to use this data for test planning because of incompatible formats, it is not captured, lack of tooling, etc.

One of the key ideas of the improvements of Visualizing Software Quality is to discover these data islands and to make the data available for the interactive visualization of software quality.

Important data sources for software quality visualization:

- Source code (Lines-of-code, Complexity metrics, test coverage by automated tests)
- Changes (code churn from the version control system)
- Defects (defect tracker)
- Test results (test suites in different test tools)

Another goal of the analysis is to gain knowledge from the history of the software project and apply it to new/ changed components. For example, correlating defects with code complexity can help to identify problems in new or maintained code. Also changed code is more likely to contain errors than unchanged.

I currently use the following criteria for metrics used in visualization:

- Interval, rational, and absolute scales (true/ false, enumeration, classifications do not work).
- Not necessarily comparable (metrics for different programming languages must be available, so that new relations can be uncovered. Even when comparing different programming languages).
- Same basis/ semantics. The metrics must be implemented in the same way for different programming languages. For example lines-of-code metric implementations come in many varieties which handle specifics differently (line breaks, statements, etc.).

6.1 Useful Metrics

Since there are so many metrics available, which ones are best? Opinions certainly differ on this topic! But we cannot do all, so I have limited the scope to a small set of what I believe are useful metrics for "Visualizing Software Quality":

- Organization - number of engineers who touched the code (indicator for potential issues regarding communication paths or know-how transfer)
- Edit frequency (number of file edits per time-frame)
- McCabe-complexity (already implemented)
- Unit-Test-Coverage/ Failed tests
- Number of bugs fixed (defect tracker fixed with revision)

Most of the metrics are discussed in Nachiappan Nagappans paper [12] which provides scientific and practical evidence on the use of metrics to improve software quality.

The “Dependencies” metric discussed in the paper will probably not be implemented for the visualizer tool because it would be difficult to implement it transparently over multiple programming languages.

To the metrics discussed in the paper I added the following to the visualizer requirements:

- Lines-of-code (Needed for visualization; already implemented).
- Performance (In most industry settings performance is still a very important aspect of software quality. It is of enormous importance that performance problems are identified right when they are created because they are very expensive to analyse and fix later.).

Risk	Metric
Organization	Number of engineers who touched the code
Development	Lines-of-code
	Complexity
	Edit frequency
Testing	Unit-test-coverage
	Failed tests
	Continuous performance

One thing about the categorization above that I do not like at all is that it will easily lead to a discussion about responsibilities and who should have access to the data, etc. I also want to emphasise the point that the separation of development and testing teams is problematic. In my opinion it is the root cause of many problems in quality assurance.

The reason why I still included the above diagram is that I want to demonstrate a tool that can help you to verify if you have a shortage of metrics in one area of risk.

Obviously, what metrics should be implemented depends on the individual project environment. There is no golden rule that says implement all of these. My advice in this situation is to think about what you can easily extract from your version control system, defect-tracker, and unit test suite and you will get a clear picture very soon. At the very least, you now have more information available than before.

7 Conclusion and future work

The paper provides a brief overview of topic of data visualization, and reference to more detailed information. The paper does not provide an exhaustive documentation on the topic of visualization, interactive visualization of software quality, or technologies used for the implementation of the visualizer tool. One way to look at this paper is as a status report of the last 12 month working on the visualizer project.

At the time of this writing, the visualizer tool implementation is far from being complete. So why provide incomplete documentation and tools to a broader audience? My intention is to reach people interested in the visualization of software quality, in order to enter into fruitful discussions on the role of software quality visualization in improving the software development process and on how to improve the visualizer tool so it can be adapted into a variety of project environments. The engaged feedback I got from conference participants at the EuroPython 2010 was already very beneficial.

The following are what are planned as the next steps for the visualizer project.

Find a solution for multivariate metrics representation. For example colourization of complex modules with low unit test coverage.

Visualizer Implementation:

- Improve data store performance
- Reduced data transfer rate between backend and frontend

- Every time the user changes one of the view parameters a complete redraw is necessary. It seems like there is some room for improvement.
- Advanced interactive user queries
- Extract data from more sources, Unit/tests coverage, Code churn from version control system, Use defect tracker to locate error prone code, Performance tests (e.g. Firefox Talos module).
- More intuitive navigation.
- Best methods for displaying testing time, especially versus test coverage
- "Data export" as a way for the user to transfer information from the visualizer to other applications. I could think of lists, todo-lists and test-case-skeletons in the appropriate formats.
- Adjustable colouration for treemaps (probably an interactive slider)

8 Acknowledgements

Parts of this paper have been presented at the annual EuroPython conference in Birmingham, UK, in July 2010. During the presentation, I discovered a huge interest in the topic of visualization and received a lot of positive feedback on the visualizer tool and how to improve the presentation to make it clearer. Since that time I have continued to work on the system and the presentation will include the most updated software version and information.

Special thanks to Richard Vireday, Intel Cooperation for the insightful discussion about the visualizer implementation and his review comments on the paper over multiple stages from the early draft on.

---

[1] Steven Johnson, The Ghost Map, 2006, Riverhead Books U.S.

[2] James A. Whittaker, Presentation on "The Future of testing" at Google GTAC 2008, http://www.youtube.com/watch?v=Pug_5Tl2UxQ

[3] Douglas Crockford, Presentation on Quality at the 2007 Frontend Engineering Summit

[4] Edward R. Tufte, The Visual Display of Quantitative Information, 2001, Graphics Press

[5] Benjamin B. Bederson, Ben Shneiderman, The Craft of Information Visualization - Readings and Reflections, 2003, Morgan Kaufmann Publishers

[6] Marlena Compton, "Visualizing Software Quality", paper and slides, 2009, PNSQC conference in Portland, http://bit.ly/c75bbo

[7] Mark Fink, visualizer tool, http://pypi.python.org/pypi/visualizer/

[8] Mark Fink, metrics tool, http://pypi.python.org/pypi/metrics/

[9] Apache CouchDB, http://couchdb.apache.org/

[10] YUI 3 — Yahoo! User Interface Library, http://developer.yahoo.com/yui/3/

[11] Norman E. Fenton, Software Metrics: A Rigorous and Practical Approach, 1991, Chapman & Hall

[12] Nachiappan Nagappan, The Influence of Organizational Structure on Software Quality: An Empirical Case Study, http://research.microsoft.com/pubs/70535/tr-2008-11.pdf

Using Static Code Analysis to Find Bugs Before They Become Failures

Brian Walker

Tektronix, Inc.
Brian.R.Walker@tektronix.com

Abstract

Automated static code analysis tools have evolved considerably from the original lint tool. They are now more comprehensive, provide more relevant results and produce fewer false positive reports. The Video Product Line organization at Tektronix has integrated static source code analysis into its nightly build process. The analysis tool has identified several errors that produced faulty behavior or crashes which could have been avoided. It has also identified subtle errors that were overlooked by human observers or simply not covered by functional testing. In concert with automated functional testing, static source code analysis has enabled the Video Product Line to achieve better quality and devote more time to feature development.

Biography

Brian Walker is a Senior Software Engineer and software architect in the Video Product Line at Tektronix. He has over 15 years of experience in embedded software development using both VxWorks and Linux. Brian started his career at Tektronix as the resident expert for DDTS, ClearCase,VxWorks and other software tools. Brian received a B.S. degree in Computer Science from the University of Colorado at Boulder.

1. Introduction

In the beginning, there was lint. Lint was developed as a tool to improve the portability of source code written in the C programming language. Lint primarily checked the syntax of a program for known weaknesses of the C programming language and provided warnings about undefined behavior, uninitialized variables and non-portable code. Later variants also validate the number and types of arguments passed to functions. These checks were useful and helped prevent a number of common defects. It is no wonder then, that many of the checks that lint performed have been integrated into compilers.

Lint, itself, is a program for static source code analysis. In short, static analysis is the process of analyzing a program without actual execution. Its converse, dynamic analysis or dynamic testing, is the process of testing a program through its execution, such as through automated regression testing, expert user testing and acceptance testing. Static code analysis also includes code reviews, but this paper will specifically focus on the use of automated static code analysis tools to improve the quality of software.

Two years ago, my department purchased Klocwork Insight, a commercial static analysis tool. We integrated the tool into our nightly builds and began performing automated static analysis in additional to our automated regression tests. The tool has provided us with a significant list of issues within our source code which has guided our efforts to improve the quality of the software.

2. Beyond Lint

The popular definition of static code analysis now extends beyond the basic lint functionality and usually includes data and control flow analysis. Static analysis can examine not just the structure of a function but also its interactions with functions that it calls and functions that call it. Rather than examine a single file, it can examine the file in the context of how it is used and how it uses data from functions it calls. A good static analysis tool can follow data through potential paths of execution and identify where it is allocated, initialized, used and eventually discarded. The analysis can be performed line by line so that each line of code is analyzed and dead code is identified. That analysis, by necessity, must be able to follow the data even as it passed by reference to other functions. When it finds a problem, the tool, can then identify the path required to produce that error.

Unlike dynamic testing, static analysis can provide nearly full coverage because it does not actually have to exercise the program to get a result. By analyzing the possible set of values of variables at each line a function, it can determine where those values would produce an error such as a null pointer dereference or an array index that is larger than the size of an array or even a negative value. The static analysis provides a list of possible errors and locations where certain possible values will produce an error.

Since the static analysis tool typically relies on a pattern library to detect errors, like any expert system, it is only as smart its teachers. Static Analysis is not perfect and will miss some bugs. And if the analysis does not go deep enough into the code, it will miss instances where the likely errors are prevented or avoided. So, the designer of a static analysis tool must balance the aggressiveness of the process for finding potential errors with the restraint to avoid false positives. So, if the tool does not produces at least a few false positives, it's probably not trying hard enough.

One of the problems with basic lint was that it would produce a large amount of warnings. The poor user, who was inundated by these warnings, might be so overwhelmed that he would simply stop using the tool or sheer number of warnings might well hide the truly significant errors that needed to be addressed. When reviewing a popular lint program, I once noted that one of the benefits of the program was that it was highly configurable and that any of its messages could be disabled in a very flexible manner. The problem with such a tool was that it required a lot of manual intervention to tailor the results. Many of the warnings were useful but having too many warnings reduces the utility of the tool.

3. Elements of a Static Analysis Tool

There are four useful features of a static analysis tool. The first, of course, is an extensive library of patterns and rules for detecting issues. It should also have a database for tracking reported issues and an interface for reporting those issues to the user. Finally, it needs a compilation process to perform the actual analysis.

The pattern library includes the analysis engine plus all of the rules and patterns used to detect errors within the software. Since not all issues are created equal, it should identify problems by severity so that users can prioritize issues, especially when the analysis is run for the first time. The library should contain patterns for a wide range of issues including potentially catastrophic errors to simple warnings. Preferably, the library also includes the ability to enable or disables patterns in the library so that users may customize working of the tool.

A database for tracking known issues should be considered an essential part of the tool. No static analysis tool can be perfect. At best, a static analysis tool provides a good approximation of the errors a program might encounter. When in doubt, the tool should error on the side of caution in order to alert users to possible problems. That naturally results in a number of false positives which, unless instructed to ignore, the tool will continue to report. The tracking database can also identify the problem more succinctly than by just the line number and continue to track the issues as the source files changes.

A static analysis tool should also have a good reporting interface so that the user can quickly determine the nature of the problem, and more importantly, how to fix it. Having the reporting interface integrated into a source code browser with a cross reference capability allows a user to investigate the problem quickly and determine the accuracy of the report. In some cases, that reporting capability might be integrated into the IDE environment.

Finally, the tool must have a function to analyze the source code and generate reports. This often takes the form of a compilation and linking process that analyzes the structures of files and combines the information with other files to produce a result. The compilation process may require as much time as the compilation and build of the executable. The tool must run the analysis with the same options and compiler flags as the normal software build so that the analysis and the build see the same lines and values within the source code.

Some have proposed that static analysis could be integrated into the compiler. There is ample reason to support tight integration of the static analysis within a compiler. For one, it absolutely ensures that the compilation and the analysis are performed on the same code. The analysis can also be more efficient by leveraging the analysis performed by the the compiler's optimizer. The analysis can be seamlessly integrated into the normal build process and the results can be displayed directly in the IDE. Both Apple Computer and Green Hills Software offer such a solution.

But, as an embedded systems developer, I first need to find a compiler that supports the target processor I am using. A tight integration of compiler and analyzer would limit my choices of compiler. Therefore, there is still a need for stand-alone static analyzers.

4. Using the Klocwork Static Analysis Tool

The process supported by Klocwork automates the generation of a build manifest by running the normal build process within a manifest generation tool that captures the build commands from the normal build script. The advantage of this approach is that it provides a simple and generally universal methods to capture the list of files and commands used to build the software. However, it does not support incremental builds so using this approach with the Klocwork tool requires either performing a full build every time or managing the build manifest separately and manually rebuilding the manifest as needed when the build configuration changes.

Since we use ClearCase, I chose the third option which is to harvest the build manifest from ClearCase configuration records. We use Clearmake to perform all of our builds which performs an audited build within ClearCase and records a complete transcript of the build including all files and commands used to build each object and assemble them into programs and packages. The format of the build manifest file is clearly described in the Klocwork documentation so I developed a simple script to extract the commands from the ClearCase configuration record and generate a build manifest for Klocwork. The use of ClearCase configuration records allows us to fully automate the static analysis process because the configuration records are maintained by ClearCase for all builds and are always complete even after performing an incremental build.

The next step is to generate a record of file ownership. Establishing ownership is not strictly required for the static analysis but it does help document responsible parties for issues that are discovered during static analysis. One way to organize ownership is by module or component where people have specific responsibility for parts of the software. In our environment, the owner of the file is simply the last person to check it in. Once again, product documentation describes the format of the file and a simple script extracts the information out of ClearCase.

Using the build manifest, Klocwork performs a build. It identifies the compiler commands through the use of a filter file which provides the names of the compilers and linker for several variants of compiler tool chains. In our case, we needed to add the names of the GNU compilers and linker because the names of the cross compilers we use are different for each target processor and often subject to the whims of the tool chain provider.

The analysis requires at least as much times as the actual build and runs in several phases. The results are collected in a table which, when complete, is uploaded to the database. The results of the build are then available through the reporting interface. This interface provides access to all of the issues recorded in the database. Klocwork provides the reporting interface as a web service allowing any user to access the results from anywhere, including my home computer.

5. Managing Issue in Klocwork

Klocwork records the state of an issue using two fields. The “State” field is the state determined automatically by Klocwork which may identify issues as new, existing, recurred, fixed or not in scope. As suggested by the states, Klocwork maintains a history of current issues and has the ability to detect when new issues are discovered and when they are fixed. The “not in scope” state means that the file containing an issues was not in the previous build. That may occur when a file is removed from a build or if a file was inadvertently not included.

The “Status” field is a state assigned by the users and reflects the priority of an issue. Every new issues starts in the Analyze state. In the Klocwork process flow, issues in the Analyze state are categorized as “uncited” with the expectation that a reviewer will cite the problem by setting the Status field to some other state. Since Klocwork is run as part of our nightly or weekly builds, I usually spend a few minutes each morning reviewing the results from the previous night. If Klocwork discovers new issues, I review the issue reports and change its status to Fix, Fix in Next Release, Fix in Later Release or Not a Problem. Issues that are in the analyze or fix state are presented as open issues and are selected by the default filters. Any user can also add a note to the issue record and Klocwork maintains a history of both the status and notes.

One of the strengths of Klocwork is the depth of the information it provides in issue reports. An issue report contains a description of the problem, usually with specific description such as “Buffer overflow, array index of ‘foo’ may be out of bounds. Array ‘foo’ of size 22 may use index values of -2 .. -1. Also, there are similar errors on lines 491, 494.” The report also provides a traceback showing all locations where a value was set or used and conditions required to get to the location where the error may occur. Due to the nature of the analysis, that backtrace will often include source lines in other functions when the value in questions is returned by a function or passed to a function. The lines referenced by the

traceback are highlighted in the source listing where they can be examined in context. Clicking on a traceback item positions the source listing at the line and highlights the line in the source listing.

The source listing also provides a cross reference interface that I can use to explore the source code. With it, I can quickly find declarations for variables, macros, constants of functions or find the implementation for a function. I can also find other places in the source code that use a class or function or are used by the class or function. The ability to quickly dive into a problem is a welcome feature of the Klocwork user interface and helps me diagnose issues or fix issues more quickly. After all, using static analysis is supposed to be saving me time and the quicker I can find the cause of a problem, the quicker I can resolve it.

Klocwork also provides direct access to the cross reference interface. That is useful for finding all of the issues found in particular files or directories. That way, if I am editing a particular file, I can use the cross reference to find all of the issues reported in a file and fix them all at once. The cross reference also provides information on the structure of the software by providing references to related objects, functions and constants. It also provides a reverse engineering capability that can help a developer understand the code.

Unfortunately, Klocwork does not integrate with our defect tracking system, or any defect tracking system. Although I can reference any issue in the database with a URL, it would be more useful if I could simply click a button and conveniently submit the issue into our defect tracking system and even more convenient if it could automatically resolve the defect in our defect tracking system. Usually, I just cut and paste the URL and send it in an email message.

6. Types of Issues Found by Static Analysis

Klocwork, like many static analysis tools, can detect several well known and preventable errors. Some of the types of problems that it detects include null pointer dereferences, array buffer overruns, memory leaks, use of freed memory, file resource leaks, concurrency violations, uninitialized data and security vulnerabilities.

Klocwork uses a convenient short hand notation to identify each problem type. For example, ABR indicates a general array buffer overrun while NNTS identifies a non-null-terminated string. The notation tends to be concise and memorable so I can often determine the type of problem in a glance. If in doubt, simply clicking on any of the notation in the web interface provides the description of the problem in the online documentation. That documentation tends to be well written and provides a description of the problem, specifies vulnerabilities, identifies risks associated with the problem and often includes instructions for mitigation and prevention of the problem.

Klocwork also provides several flavors of issues. One common distinction are the classifications of “must” and “might” which indicates whether a potential error depends on one or more factors. A “must” issues indicates that the error will occur as a results of a single failure. A “might” issues indicates that a failure might occur if more than one condition is true. For example, an NPD.FUNC.MUST issues describes a situation in which a pointer, returned by a function that could return NULL, is always dereferenced by the caller. An NPD.FUNC.MIGHT issue describes a situation in which the caller might dereference that pointer if certain other conditions are also true.

Klocwork has been very effective in discovering null pointer dereference issues within our source code. Many of these issues are the results of source code that does not check the values of pointer parameters passed into a function or the pointers returned by a function. Klocwork can identify several varieties of null pointer dereferences depending on where the pointer dereference occurs and whether the program actually checked the pointer. For example, NPD.CHECK.MIGHT describes an issue in which a pointer was previously checked for NULL but the program dereferences the pointer anyways. This problem can be quite insidious because it can easily be missed in a code review. Reviewers who are not paying attention may see the check for NULL but miss the nuances of the implementation. In the following sample, the pointer pdu is checked for NULL but only if reqid != state->reqid.

```
int handle_request( int op, int reqid, PduType *pdu, SyncState *state)
{
   if (reqid != state->reqid && pdu && pdu->command != MSG_REPORT) {
      return 0;
   }

   if (op == RECEIVED_MESSAGE) {
      if (pdu->command == MSG_REPORT) {
```

There are also several flavors of array buffer overflows and Klocwork has discovered quite a few in our source code. String handling is a common theme in the issues reported by Klocwork such as using strcpy() to copy strings of different or unknown sizes. One interesting scenario that Klocwork has reported involved the use of sprintf() to construct a string. Klocwork reported an issue because the programmer had sized the buffer to fit the expected result, but not large enough to handle unexpectedly large numbers or long strings. Another common error involves iterating through an array using the wrong constant as an upper bounds for the array index.

Most memory leaks are of the MLK.MIGHT variety and are often caused by improper error handling. The typical scenario involves the handling of errors conditions where the code returns from a function without freeing memory that was previously allocated by the function. These sections of code are typically not covered by normal functional tests because the errors they handle are not encountered under normal conditions. Unit testing might cover this sort of failure if they were written and if the programmer was diligent about testing all paths through a function. Memory leaks can also be difficult to detect because memory is often lost a little at a time. That may not matter in an application running on a desktop with massive amounts of virtual memory at its disposal. But, if your goal is 24/7 operation in an embedded device, those memory leaks tend to add up to produce subtle failures and memory fragmentation.

Another place that Klocwork looks for memory leaks are in class object initialization. If a C++ class allocates memory in the constructor, Klocwork will verify that the memory is freed by the destructor. Likewise, if memory is deallocated in the destructor, Klocwork will warn of the potential for double freeing of memory if the class does not define both an assignment operator and a copy constructor.

Klocwork includes several checkers for use of memory after it has been returned to the system heap. Prior to running Klocwork, I did not expect to see this particular issue in our software. Surprisingly, Klocwork has reported this issue in third-party and open source software that we use in our product.

File resource leaks, like memory leaks, are often caused by improper error handling but the pool of file handles is more limited. However, resource leaks are given a lower severity by Klocwork. While they are reported as more of a warning, than as an error, Klocwork has found a number of locations in which file handles could be lost.

Since our software is multithreaded, the ability of Klocwork to detect concurrency violations has been useful and has uncovered problems where semaphores could be acquired indefinitely. Klocwork provides warnings when semaphores are taken within a function but not unlocked when the function returns and in situations where a thread goes to sleep while a semaphore is locked.

I expect that any static analysis tool should be able to detect use of uninitialized data. Often, it appears, people simply forget that automatic variables within a function are not initialized to a specific value.

Security vulnerabilities are becoming a much greater concern as more devices become connected. However, secure programming practices are less obvious and easily overlooked. A vulnerable program will still pass it's functional tests and the vulnerabilities will not be exposed until someone specifically tests for them or tries to exploit them. The number of reported vulnerabilities in software continues to grow and the cost to fix them in the field may not only include the time and expense to fix the defect but also carries the potential for exposure of sensitive data and erosion of customer trust. Most vulnerabilities can be traced to a relatively small number of common programming errors (Mitre/SANS 2010). This an area

where prevention is far more effective than remediation and regularly scanning for these common errors can reduce or prevent most security vulnerabilities.

Many of the issues in the Klocwork pattern library can be mapped to the Common Weakness Enumeration dictionary sponsored by the National Cyber Security Division of the U.S. Department of Homeland Security (Klocwork 2010). Klocwork analyzes code for several specific types of security vulnerabilities including unvalidated user input, SQL injection, path injection, cross-site scripting, information leakage, weak encryption and other vulnerable coding practices. Unvalidated user input can cause a whole host of issues including buffer overruns, injection of executable code, spoofing and compromised programs.

Klocwork recently introduced support for the MISRA C and C++ coding standards. The MISRA C and C++ standards were created by the Motor Industry Software Reliability Association to facilitate safety and portability of software written for automotive control systems. These standards have also been adopted by developers in the aerospace, telecommunications, medical devices, defense, railway and other industries (MISRA 2009). The standards define a coding standard composed of over a hundred rules to make software more reliable.

7. Analyzing the Results

After running the Klocwork tool on our code base of about 500,000 lines of C and C++ code, the tool produced a list of about 1500 issues. After reviewing those issues, I eventually rejected about 175. The remaining issues were significant problems and, in some circumstances, blatantly incorrect. In my judgement, the total number of issues reported and the number of false positives were quite reasonable.

Unfortunately, the problem with reporting 1500 issues in an established code base is that someone must go through those reports, categorize them and actually fix the problems. Introducing a static analysis tool to an established code base will most likely create a sizable bug debt. However, once the bug debt has been paid, the true value of static code analysis will be that the problems found can be quickly identified and corrected even before they are tested or delivered to customers. Static analysis is most useful when it is employed on new new software as a preventative measure.

There have been instances where I have literally spent hours trying to reproduce and capture an instance of a crash bug with a debugger and then found that Klocwork had found that same bug, identified its location and the conditions necessary to produce the crash. In one case, the crash was due to a race condition that was only triggered under certain specific conditions and in a particular order. I spent most of that time trying to reproduce the failure hoping to catch it in gdb. Had I checked the Klocwork database and fixed the bug as soon as it was reported, I could have avoided the grief.

Other bugs express themselves in less subtle ways. Most of the time, when I have encountered bugs that are the result of a stray pointer, the cause tends to be a simple array buffer overrun. Sometimes, the array just happens to be in memory adjacent to a data structure that is used by a different thread so using a debugger to step through the code does not always find the culprit. In some cases, the stray pointer writes over a virtual table causing the program to crash mysteriously. Finding such a bug requires first finding the location that gets overwritten and then setting a watchpoint to trap the write. Finding it with static analysis can literally saves days of aggravation.

A substantial number of the issues found by static analysis were in error handling and were missed by functional testing precisely because they do not normally occur in the normal execution of the application. Some would have been caught by unit testing had the programmer written a test to stimulate that path. Unfortunately, the error eventually does occur and instead of gracefully recording the error and continuing, the application would unceremoniously crash. Hopefully, that moment of failure occurs before the software is delivered to customers.

Another advantage of static analysis tools is that they also promote good programming practices. Many of the issues discovered by static analysis go back to basic principles like check those pointers for NULL

before dereferencing them, check your array index before using it to access an element in an array or close that file handle before returning an error from the function.

8. Looking Forward

Currently, my team has only integrated static analysis into the nightly builds. The results are available on a daily basis. An even better integration would be to integrate the static analysis into our development environment and provide direct and immediate feedback to all of our developers. Klocwork and other vendors of static analysis tools offer integrations for both Eclipse and Microsoft Visual Studio.

Conceptually, we would continue to run nightly builds to perform the bulk of the analysis. The client tool could then link to the shared database, run the analysis on the files currently checked out in the user's sandbox and cross reference the information to provide results for the software as it is currently configured in the user's environment. That capability would allow the user to receive immediate results before committing changes to the version control system. Eventually, the static analysis could be used as a gate to checkin of the files into the version control system and thereby prevent coding weaknesses from leaving the developers desktop.

9. Fitting it all Together

Static analysis is an excellent tool for finding problems in source code early in development. The ability to analyze code without execution enables software to be analyzed before it can be loaded and run on an actual hardware. It can provide more complete coverage of the source code than dynamic testing for a finite set of known problems because it is able to follow all paths within each function without reliance on external stimuli. Testing and analysis are adept at finding different kinds of of problem. For one, static analysis cannot determine if the operation of a software program conforms to the product requirements. And, testing cannot accurately or repeatedly determine the structural soundness of the software.

Likewise, static analysis can complement the practice of formal and informal code reviews. While human reviewers can make mistakes and overlook problems, static analysis can consistently detect a known set of common weaknesses. It's kind of like having an older brother who is always willing to tell you about your faults. But static analysis cannot replace expert eyes of peers who not only have the imagination to look beyond the code but also have an understanding of the product requirements and the experience to offer better solutions.

Sometimes, there really is no other option. Most software development uses open source and third party software. Short of assigning an engineer to review all of the code or write tests to thoroughly test the code, how does one assure that the software is free from defects? Some open source communities invest in test suites but many do not. Most open source software is provided "as is and without warrantee" according to the terms of the GPL or other open source license. In my team's product software, more than half of the remaining issues in the Klocwork database were detected in an open source library.

Automated static analysis is not a silver bullet but it can provide immediate feedback to help write better software. It can increase the tribal knowledge of an organization by introducing a knowledge base of known software weaknesses and can provide information on how best to avoid them. If used frequently, it can allow an organization to find and fix those problems before testing and before they become bugs that annoy customers.

References

MITRE/SANS 2010, “CWE/SANS Top 25 Most Dangerous Software Errors” MITRE Corporation, http://cwe.mitre.org/top25/index.html (last modified June 29, 2010)

Klocwork 2010, “Detected C/C++ Issues” Klocwork Inc., http://www.klocwork.com/products/documentation/Insight-9.1/Detected_C/C%2B%2B_Issues (last modified May 18, 2010)

Klocwork 2010, “Detected Java Issues” Klocwork Inc., http://www.klocwork.com/products/documentation/Insight-9.1/Detected_Java_Issues (last modified April 16, 2010)

MISRA 2009, “MISRA C” MISRA Consortium, http://www.misra-c.com/Activities/MISRAC/tabid/160/Default.aspx (accessed August 15, 2010)

Coverity, “Resource Library, Case Studies” Coverity Inc., http://www.coverity.com/html/research-library.html#CaseStudies

Various Authors, “Static Code Analysis” Wikipedia, http://en.wikipedia.org/wiki/Static_code_analysis

Adaptive Application Security Testing Model (AASTM)

Ashish Khandelwal (Ashish_Khandelwal@McAfee.com**)**
Gunankar Tyagi (Gunankar_Tyagi@McAfee.com**)**

Abstract

Security testing is synonymous with terminologies such as Cross Site Scripting (XSS), SQL Injection, Key logging, backdoors, phishing attacks, and so on. However, the lack of skill & experience in Application Security Testing prevents practical implementation of security testing, and most project teams do not know where or how to start.

This paper follows the experimental ride of our team of functional testers and the new approach towards Application Security Testing. As a team, we faced many challenges from the beginning in understanding Threat Modeling, struggled to implement the same in a time-constrained manner, and had to deal with the failure to finally implement and yield value. We subsequently created a self-adjusting model, which we have coined as "***Adaptive Application Security Testing Model***".

The Adaptive model aims to maximize the efforts an Application Security tester puts in. With the hard deadlines and resource optimization that are commonplace in the industry, this paper gives you a strategy to achieve Application security without compromises.

Here are some of the strategic points, which an Application security tester can leverage from our paper:

a) How to position themselves on our Adaptive Ladder
b) Understand Application Security from a rudimentary yet adaptive perspective
c) How to create Attack Models
d) Review real case studies

Biography

Ashish Khandelwal *has more than 5.5 years of Software Testing experience. He holds a B.Tech degree from IIT Kanpur and works as a Senior QA Engineer with McAfee Enterprise product solution group. Being a Certified Ethical Hacker from EC-Council, he is interested in the latest Security Testing trends, and continuously seeks to improve software security. Ashish works towards becoming a technology solution consultant/ architect by providing technical insight to different process verticals in testing solutions.*

Gunankar Tyagi, *a Computer Science Engineer has around 2 years of experience solely in Black Box testing. Being CompTIA Security Plus certified, he has earned many accolades for his out-of-the-box testing skills and has proven himself a respected tester in a very short time. His areas of interest delve more into Security Testing.*

1. Introduction

Have you ever thought how easy it would be to write simple test cases that can reveal the security loopholes in your product? However as many of you may have experienced, this is easier said than done. We ourselves have witnessed quite a few of the security testing initiatives abandoned midway. We usually feel the need to justify it in terms of why we need it, do we need it at all and what will we achieve from it? After all, this is the herculean effort of bringing the security-testing framework alive. Even though the proven "Threat Modeling approach" remains the best tool available, it seems to be too expensive in terms of energy and effort investment

This paper tells our story as a test team, as we took the initiative to conduct security testing on our product. We attempt to describe the various challenges we came across and how we weaved our way around them.

We were not successful in finding security defects in the early part of our testing but soon we found that the very nature of 'Threat model approach' made it difficult and discouraging for beginners like us. The real breakthroughs started happening only when we realized that threat model itself is the hidden barrier and that we would have to reevaluate our strategy to move ahead. This led to a new approach and we have named it Adaptive Application Security Testing Model (AASTM) model.

We would like to begin by answering a few basic (frequently asked) questions to explain the need of security testing in general. Then we would like to share a few challenges faced by us in implementing application security testing. After this, we will elaborate the two-tier AASTM approach and its security testing types with the help of case studies. We conclude the paper with an Application security-testing checklist and a few lessons we learned along the way.

2. Application Security Testing – Primary Needs

In this section, we try to answer some of the basic questions, which come into the mind of security tester when he embarks with breaking the product.

Why to do security testing?

Companies discredit the need for Security testing because they feel there is no return on investment and thus most of the time the need of security testing is not taken seriously. Management does not always know about product flaws; company director assumes that every function works smoothly without any defects. However, experience shows that no product/ system can be deemed completely secure without controversy. There will always be bugs in a program; whether they are found or not is another question.

The losses associated with security flaws have been heavy as can be analyzed from the following security incidents:

TJX Company Breach

The TJX Company breach, which was first reported in January of 2007, has been widely recognized as the largest reported theft of personal details ever lost by a company.

Monster Job Site Hacked

Monster.com suffered a heavy security breach in Aug 2007 that reportedly resulted in the theft of the confidential information for some 1.3 million job seekers. Hackers stole information from the US online recruitment site's password-protected CV library by using credentials taken from Monster clients. They launched the attack using two servers at a web-hosting company in the Ukraine, combined with a botnet.

Operation Aurora

In mid-2009, the most ultra-sophisticated attack was uncovered for which McAfee later coined the term Operation Aurora. Reportedly, Chinese hackers have used a previously undiscovered security hole, existent in several major versions of Internet Explorer. This attack affected as many as 2,411 companies and compromised data ranges from intellectual property, classified documents to credit card transaction details. Such Advanced Persistent Threats are expected to grow in the future

Microsoft and the Love Bug

The love bug, also known as the "I LOVE YOU" virus was made possible because the Microsoft Outlook e-mail client was (badly) designed to execute programs that were mailed from possibly untrusted sources. Apparently, nobody on the software team at Microsoft thought through what a virus could do using the built-in scripting features. The damage resulting from the "I LOVE YOU" virus was reported to be in the billions of dollars.

Today, the hackers have organized themselves and hit the market at a time to cause maximum loss. To make matters worse, the tools available to the attackers are becoming more sophisticated and easier to use. Clearly, a product with vulnerable loopholes has a high probability to bring bad name to the company and thus a prime need to perform security testing.

How to do security testing?

This is where most of the initiatives are killed off, since no one has a clear picture of how to do it, and thus, the whole procedure gets undefined and unclear. The most general approach is to follow Threat Modeling. Nevertheless, is this the right way for you? Could some other strategy help you perform security testing effectively? We will address some of these questions as the paper follows.

Who will do security testing?

Certainly, there are no pre-requisites for a tester who want to perform security testing. The right amount of attitude with zeal to learn could be the starting point for a security tester.

We would like to introduce you to our product environment. We are a team of 15 QA Engineers and 20 Developers distributed across multiple location. The skill-sets and experience levels varies across the team, ranging from Software Engineers working in Quality to QA Analysts with deep domain knowledge. As an initiative, we (a team of 2 functional testers) took up the challenge to find security loopholes in our product. The next section describes the threat model with which we started and gives an idea as to why we failed with this model.

3. Getting Started – Threat Modeling

What Is Threat Modeling?

Threat Modeling is a structured approach to identify, quantify, and address security threats in the application. A threat can be viewed as a potential or actual adverse event that may be malicious (such as a denial-of-service attack) or incidental (such as the failure of a storage device), and that can compromise the assets of an enterprise.

The basis for threat modeling is the process of designing a security specification and then eventually testing that specification during application design.

Threat Model Process

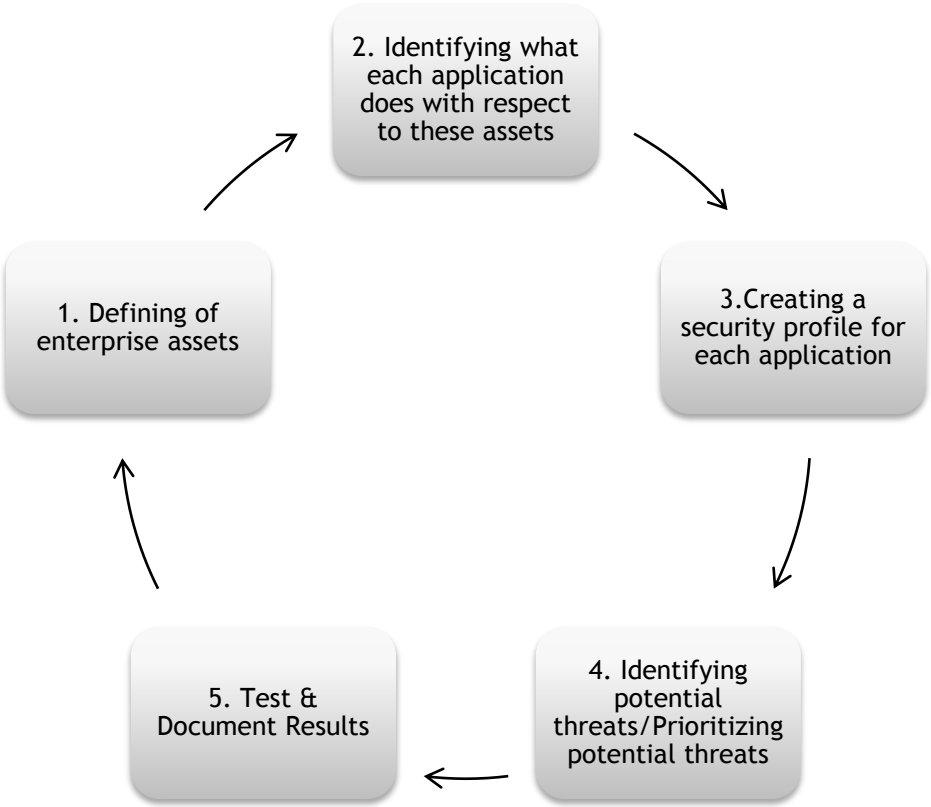


Figure 1 Workflow of a Threat Model

4. Threat Model – Trinity of Constraints

Though the model seems easy to comprehend, it is fairly difficult to implement. As we tried to follow this traditional way, we encountered three constraints as described below:

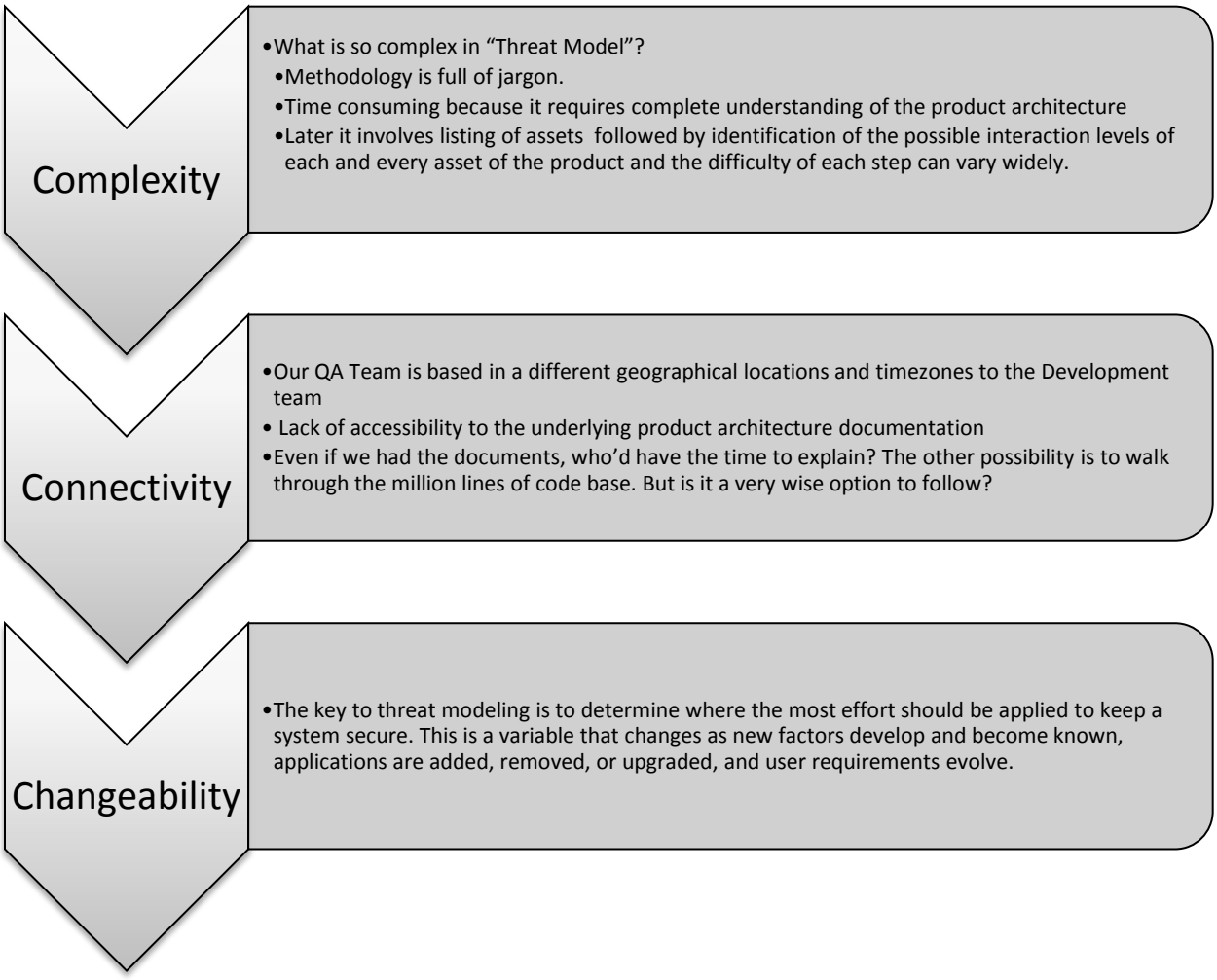


Figure 2 Trinity of Constraints

Finally as we found our motivation sagging we had two choices – either to quit or to find another way. We decided against quitting and decided to step back and reframe our strategy to see if we could come up with something useful. We came up with a strategy that we named AASTM model.

5. AASTM Model – Adaptive Ladder

"Adaptive" approach focuses more on finding security defects while the expertise level is still building. The ladder shown in Figure3 provides the foundation of this approach and compares it with traditional threat model behavior.

As we move up the ladder, the expertise of the security tester increases and gives us an edge by finding early defects that could be functional as well as security. Each step up the ladder optimized our results, but in turn required consistent upgrading of security testing skills and product knowledge.

This ladder also compares the two models in terms of the results achieved as represented by the colored circles (R1, R2, and R3). In Threat Model approach, we would have to go through the entire cycle to uncover the security defects. Although our approach provides the same result at the end of the cycle but with it you can start uncovering the security defects one by one as you step up the ladder. The significance of the security defects found increases as we move up the ladder and this strategy ensures that the tester involved does not lose motivation.

As can be seen from the ladder, it is a two-tier approach broken down into "*Peripheral* & *Adversarial*" described later in this paper. Adaptive ladder gives a foundation to this model that we named as "Adaptive Application Security Testing Model".

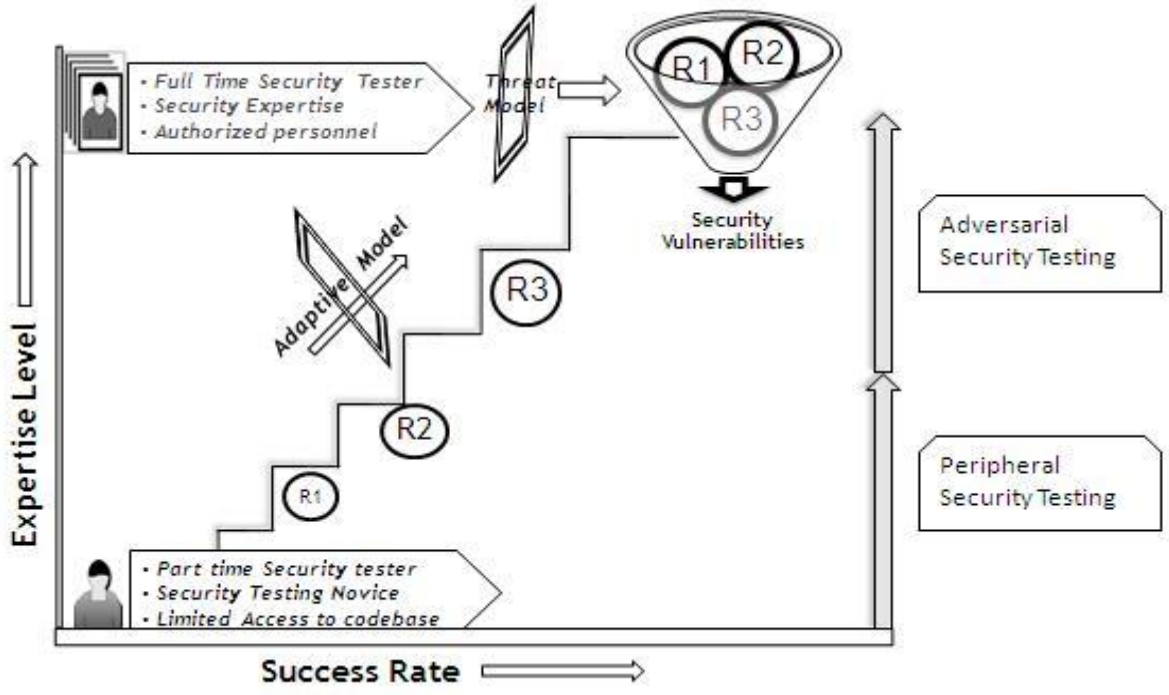


Figure 3 Adaptive Ladder

6. AASTM Model

The “Adaptive” Model in its initial stage is centered more on stimulating the security testing approach by focusing on finding early security defects rather than on studying deeply into technology and product in order to attack it. It increases enthusiasm and adaptability in a complex security arena. As we moved along with this approach, we started enhancing our knowledge and constraints to maximize our potential and results.

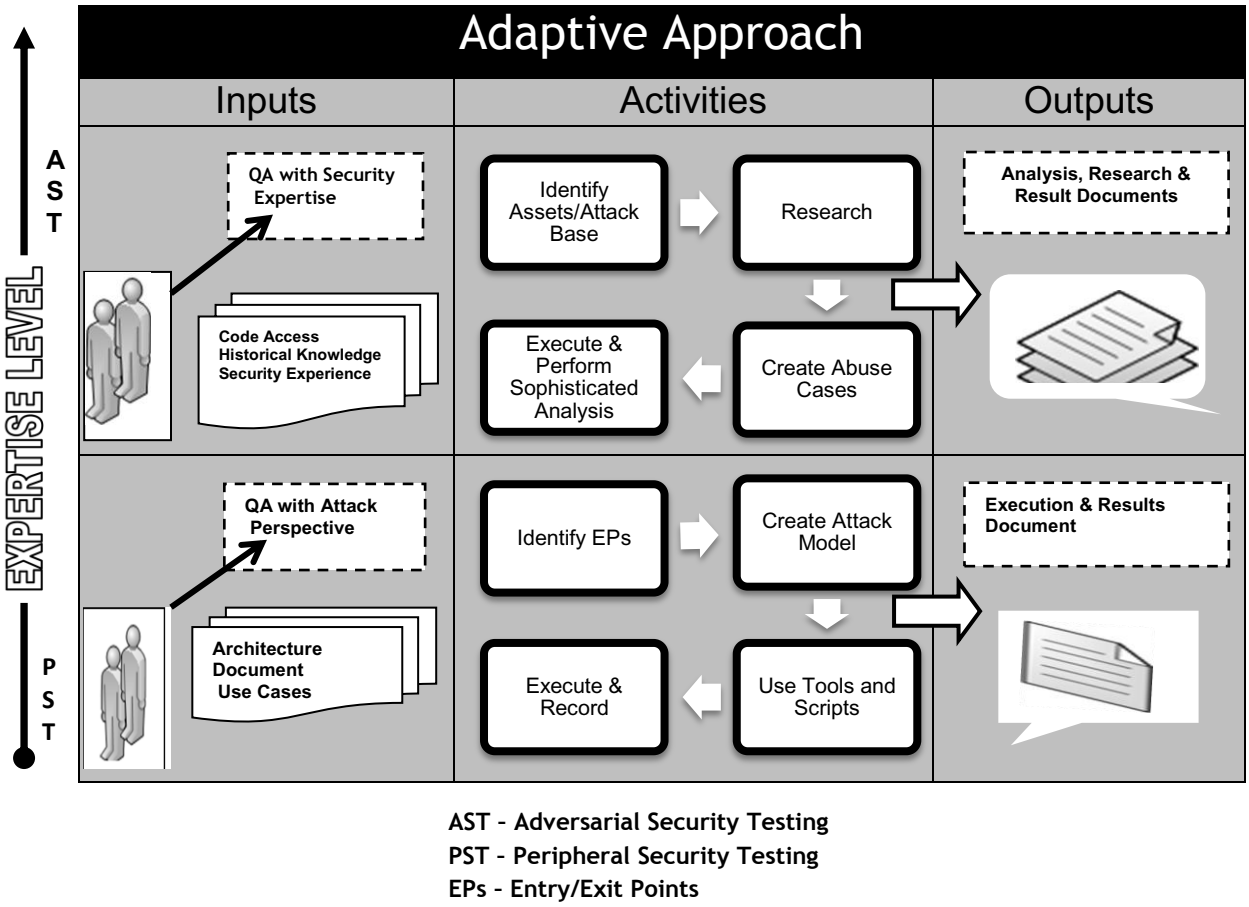


Figure 4 Adaptive Model

As a two-tier testing approach, this model talks about Peripheral and Adversarial Security testing. Adversarial testing is a successor of Peripheral. As we go up the ladder, our methodology changes and hence our priorities to test change. Each testing type is classified in terms of Inputs, Activities and Outputs as seen from *Figure4.*

- **Inputs** are the prerequisites required in order to conduct the particular type of testing.
- **Activities** describe the flow to perform the particular type of testing.
- **Output** takes care of results and analysis.

6.1 Peripheral Security Testing

As a team of functional testers, we were initially following the traditional way of doing security testing, by using Threat Models. However as time passed by with no results, we shifted our focus from traditional method to result based approach. We followed a peripheral approach that helped us in finding the low-hanging fruits together with building security expertise.

This is an entry point based, Black- Box testing approach with the aim of quickly finding surface-level but critical security issues.

Some of the key-definitions involved in this type of testing include:

Entry Point

An entry point is a place where inputs are supplied to your application. E.g. Files/ Folders/ Application UI

Exit Point

Any desirable/ undesirable output from the application. E.g. Log files/ tmp files

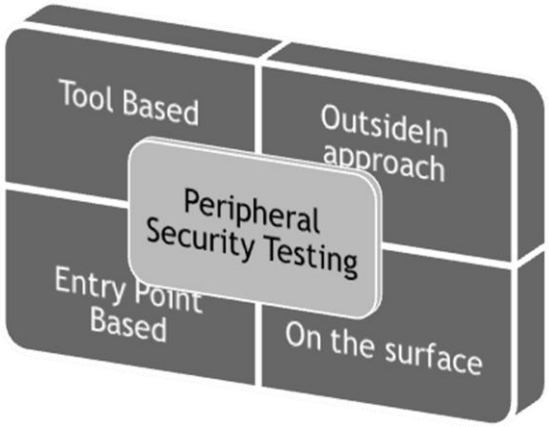


Figure 5 Peripheral Security Testing

Outside-In Approach

This technique breaks the software from outside without knowledge of internal implementation and thus enables testers to find security loopholes the black box way.

On The Surface

The targeted issues are easier to detect and require less effort.

6.1.1 Case Study I (Peripheral Security Testing)

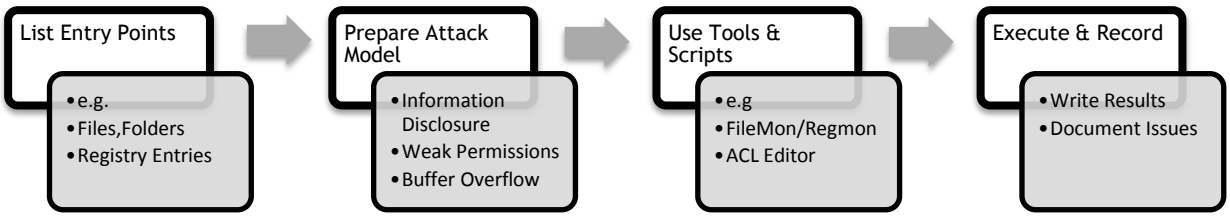


Figure 6 Case Study I (Peripheral Security Testing)

This is a 4-step process as briefed in the figure above. After identification of the entry points, the tester needs to prepare an attack model that is supposed to break the software. Lastly, we need to run the attack scenarios using appropriate tools and perform the result analysis.

List Entry Points

One of the easy ways to find defect is to work upon the most commonly used components of your software. Files, Folders & Registry Entries are notably the most essential and populated components of a product. We aim to explain here the exploitation of these components by following Peripheral approach.

Prepare Attack Model

Here are some preliminary questions, which leads to the preparation of this Attack Model.

Attack Model Development

- ✓ *Does your product functionality hamper if you deny the permissions to the temp folder?*
- ✓ *Do the files (logs/event xml/binaries) contain sensitive data?*
- ✓ *Is there is a way in which you can cause buffer overflow in the file extension /file names?*

This is not the end of the list; it may vary depending upon the tester's thought process and the application knowledge.

Each of the points in Attack Model development leads to one or more attack scenarios. The next section utilizes this attack model and chooses appropriate tools to crack the application under test.

Use Tools and Scripts

SysInternal Tools like RegMon, FileMon and ProcMon are used to monitor real-time File System, Registry, Process or Thread activity. By defining appropriate filters, we can monitor the resources of the product under test. The output of these tools could be analyzed to find information disclosures or Buffer Overflow conditions.

Other built-in Windows Tool called ACL Editor can be used to verify or manipulate the object permissions to detect weak permission flaws.

Few of the times, analysis of the result of one tool gives a way to attack the product using another tool. For instance, once RegMon discovers that our product uses an exploitable Registry Entry, we could find a way to manipulate the permission of this object using ACL Editor.

Execute and Record

The last step is to utilize the tools potential to find product loopholes with the aid of attack models. Side by side, we record our results for the issues found. In addition, the details of the results could be helpful for developing attack models and increasing knowledge base for our product.

Let us assume that our product uses temp folders extensively for doing file system operations. Here, the security issue could be grave if we leave some confidential information inside the temporary folders or if denying permissions to the temp folder makes our product non-responsive. At the same time, we can utilize this knowledge to perform buffer overflow if we repeatedly create tmp files using automation script.

Please refer **Appendix A** for a comprehensive list of various entry points with their attack models and tools list.

6.2 Adversarial Security Testing

As we uncover the surface defects using peripheral approach, it gives us the ability to think maliciously, improvise knowledge base and increase expertise level. After scrutinizing the entry points applicable to our product, we can aim to use our acquired knowledge in order to find the hidden security loopholes.

We achieve this by streamlining our strategy with the Adversarial approach. It is an expertise-based, hostile Security testing approach, which is carried inside out to reveal loopholes in the product. As a successor to the peripheral approach, this enables the security tester to attack the product in holistic way. It takes historical knowledge and security experience as the inputs with the target on the hidden security loopholes.

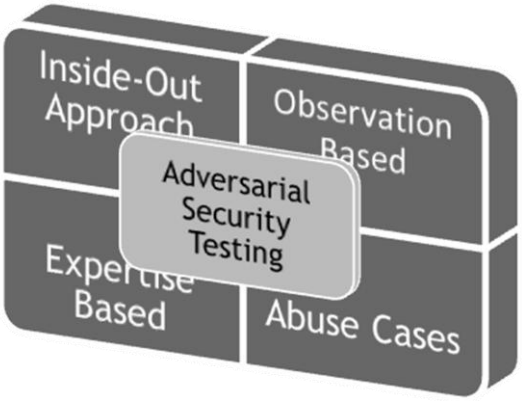


Figure 7 Adversarial Security Testing

Here are few of the key-definitions required in order to understand this approach.

Attack Base

An entity of the product or the Operating System, which can be manipulated to perform an attack on software.

Inside out

This technique requires studying the product internals and eventually uses the data or control flow of a program to break the software from inside.

Historical Knowledge

Gather past vulnerability information about the attack base. Have a quick check in the vulnerability repositories like *cve.mitre.org, securityfocus, osvdb* etc.

Abuse Cases

Abuse cases (sometimes called misuse cases as well) are a tool that can help you begin to think about your software the same way that attackers do.

6.2.1 Case Study II (Adversarial Security Testing)

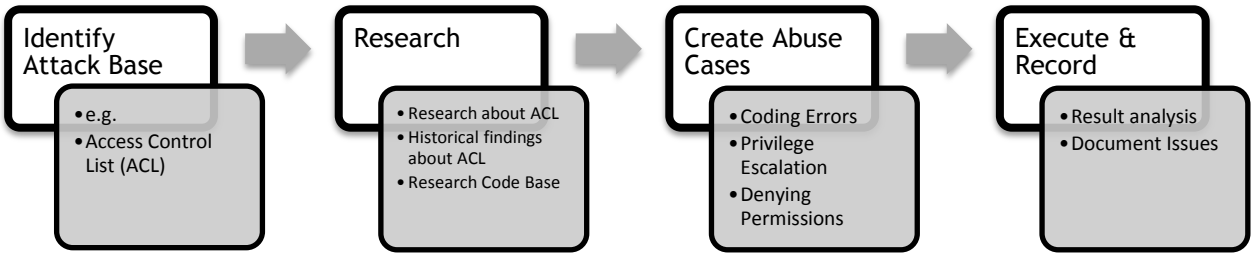


Figure 8 Case Study II (Adversarial Security Testing)

Identify Attack Base

Windows ACL is quite literally your application's last backstop against an attack, with the possible exception of good encryption and key management. This is fundamental part of Microsoft Windows NT and later, however, to some extent one of the least understood feature of Windows. It is not always possible for the developers to map permissions with the resources due to various reasons such as ignorance, design issues or high cost.

Being a fundamental component of Windows, it provided us a good case study by exploring the Access Control Mechanism used by our product. We will follow a 4-step adversarial approach to explain the exploitation of this attack base.

Research

After identifying the attack base, our task is find product ACL weaknesses by applying historical knowledge and analyzing code base.

To start with, we can perform code coverage of our product and look for any of the following improper ACL definitions types

a) NULL ACL Definition

b) Dangerous ACE types

c) Verify if proper Access Control Lists are defined for the product resources

Vulnerability disclosure websites such as CVE, Secunia or SecurityFocus are the major source of gathering historical knowledge about any Attack Base. Until Windows XP, Shatter Attack was one of the most prominent attacks to exploit Windows ACL. It takes advantage of a design flaw in Window's message-passing system where arbitrary code could be injected into any other application or service running in the same session. If your application has a graphical interface, which runs with higher Local System privileges, possibly it will fail with Shatter Attack.

Create Abuse Cases

Now, based on our research the following defined the foundation of Abuse Cases development.

```
Abuse Cases Development
-----------------------------------------------------------------
✓ Complexity of ACL's configuration
✓ Permissions cannot be assigned to all objects
✓ Exploiting Integrity Level (Windows Vista/Windows 7 specific)
✓ Shatter Attack
-----------------------------------------------------------------
```

Depending upon individual approach and product behavior, above list can be manipulated.

Execute & Record

A part of the execution is completed in the research section where we looked for improper ACL definitions in the code base. In addition, we can perform shatter attack on the product components or try to exploit Integrity Level.

Finally, we report the issues and document the results found by exploiting ACL.

For more information about this testing type, please refer to **Appendix B**, which displays the list of various attack bases together with their Abuse Case scenarios.

7. Conclusion

No doubt, in absence of a guided test scenario and lack of product management support, performing security testing becomes a tough task. It goes via myriad motivational issues, as success is not expected to come overnight (not even over months). To make security testing a part of the functional Black-Box testing, one would need to create the necessary skill-set first, which can be appended only by technical certifications, user group studies, basic understanding of the security concept, network/OS elements.

In today's market, increase in the number of security incidents has made security testing an inevitable part of SDLC. Hence, treating security testing as any other testing type, rather than continuing to give it a specialized treatment, would be a good start. Adaptive approach promises to empower functional testers to perform security testing and thus brings a security solution easy to implement and adopt by the management.

We hope that our model simplifies the understanding of application security testing and optimize the efforts put down by testers in finding security loopholes.

8. References

a) *Tom Gallagher, Bryan Jeffries and Lawrence Landauer , Microsoft Press Hunting Security Bugs, 2006*

b) *G. Hoglund and G. McGraw, Exploiting Software, Addison-Wesley, 2004.*

c) *http://news.cnet.com/2100-1001-240112.html*

d) *http://en.wikipedia.org/wiki/Operation_Aurora*

e) http://www.infosecwriters.com/text_resources/pdf/need_for_security_testing.pdf

f) http://www.zdnet.co.uk/news/it-strategy/2007/11/14/the-worst-it-security-incidents-of-2007-39290745/

g) http://en.wikipedia.org/wiki/Shatter_attack

h) http://en.wikipedia.org/wiki/Mandatory_Integrity_Control

i) http://archive.hack.lu/2007/cracking_windows_access_control.ppt

Appendix A: Peripheral Security Testing checklist

Table 1 Peripheral Security Testing Checklist

S.No.	Entry/Exit Points	Attack Model	Tools/Scripts
1	File & Folders	Information Disclosure Weak Permissions Buffer Overflow	FileMon ACL Editor strings
2	Sockets	Man-in-the-middle Attack Sniffing network traffic Send malicious data	Wireshark netstat.exe netcat
3	Registry Entries	Registry Accessed by the product Permission of the registry keys	Regmon ACL Editor
4	Named Pipes	Exploit weak permission Hijack the creation Impersonate the client	PipSec PipeList CreateAgentPipe ObjSD
5	User Interfaces	Shatter Attack Format String Attacks	Shatter Tool WebText Convertor
6	Command Line Arguments	Exploit Undocumented command Line switches	Command Line switches /?, -?, /h, or -h. Process Explorer Image tab
7	Environment Variables	Uncover Environment Variables used by Product Manipulating data inside Product defined Environment Variables	Process Explorer Environment Tab System Environment Variable Tab
8	ActiveX Control	ActiveX Repurposing Attacks ActiveX Fuzzing	COMRaider OLEView
9	Drivers	I/O Verification DeadLock Detection Dangerous APIs Exceptions/Handlers/Memory Pool Tracking Loading and Unloading Filter Driver Attach and Detach Filter Driver	Windows Utility-> Verifier.exe Windows Utility -> fltmc Microsoft Application Verifier Velocity Tool by Microsoft

Appendix B: Adversarial Security Testing checklist

Table 2 Adversarial Security Testing Checklist

S.No.	Attack Base	Abuse Case Scenarios	References and Historical Knowledge
1	Access Control List	Verification of apt ACL's for your product resources Target NULL DACL Look for dangerous ACE types -> Everyone (WRITE_DAC) -> Everyone (WRITE_OWNER) -> Everyone (FILE_ADD_FILE) Target Windows DAC weakness Target Windows MIC weakness	Shatter Attack http://www2.packetstormsecurity.org/cgi-bin/search/search.cgi?searchtype=archives&counts=26&searchvalue=win2000+attack+.c Exploiting Integrity Levels http://archive.hack.lu/2007/cracking_windows_access_control.ppt
2	Shell Extensions	List out shell extensions used by your product List the resources utilized by our Shell Extension Behavior of shell extension. Effect of impersonating your product shell extensions.	http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-5902
3	Plugins	List resources used by BHO Learn how to write a BHO Understand functionality of IE Plugin. -> This can give more attack vectors Effect of impersonating our product BHO Find a way to disable IE Plugin	http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2004-2382
4	Denial Of Service	Do basic analysis of DOS Attacks Identify the services rendered by your product Identify the ports used by the services Identify tools to send specially crafter packets to perform a DOS Attack on our product. (use historical info) Analyze the results	http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1855

Large-Scale Integration Testing at Microsoft

Jean Hartmann

Test Architect, Developer Division Engineering Systems

Microsoft Corp.

Redmond, WA 98052

Tel: 425 705 9062

Email: jeanhar@microsoft.com

Abstract

Given the large number of components and dependencies, substantial time and resources are expended by Developer Division teams in integrating and validating these components into the product. In past releases, this flow of code between product branches was often hampered by component teams inadequately qualifying their code contributions in these branches during reverse integrations (RIs) and compounded by the fact that important dependencies were being taken on their code, often without their knowledge. As a result, when those dependent component teams merged those unstable code contributions into their own code base, that is, product branches as part of their forward integrations (FIs), breaking changes proliferated with respect to product and test code - important integration points had not been exercised by the RI'ing team. All teams subsequently paid a heavy price in terms of failure analysis. With this scenario repeating itself numerous times during the development cycle, dependent teams became wary of taking FIs and code velocity began to slow. This resulted in a product, which was not updated often enough and many times ended up in an unstable state.

In this paper, we describe an ongoing test improvement initiative, which supports efforts to reinvigorate this code flow, and ensures that code quality is not compromised. We more closely examine some of the past issues that led to the stagnating code flow and describe our new approach to integration testing by focusing on the new strategy, processes and tools.

Biography

Jean Hartmann is a Principal Test Architect in Microsoft's Developer Division with previous experience as Test Architect for Internet Explorer. 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

As the size and complexity of flagship products, such as Visual Studio 2010, continues to increase, the number of challenges surrounding product code integration and its validation are also rising. These challenges are technical and logistical in nature, yet typical of any large-scale software development. In our case, we have a large number of individual teams working around the world on different parts of the product in so-called *product branches*. They contribute their code components and test collateral at regular intervals via *Reverse Integrations* (RIs) to the existing code base for the entire product. Conversely, teams regularly absorb that newly integrated code via *Forward Integrations* (FIs) into their product branches. Thus, a code flow is established for the product and ultimately enables the product to be shipped with the required features.

The key challenge that we face involves *improving code velocity* with RIs and FIs occurring on a more regular and frequent basis. Whenever teams, that is, product branches are broken by flawed code changes propagating throughout the product (build tree), everyone has to expend significant resources fixing test and/or product issues. Depending upon the number and frequency of such breaking changes, teams become reluctant to take FIs and in turn, initiate fewer RIs. They, in effect, start to isolate themselves more and more in order to retain their development momentum within a branch. The result of this growing isolationism is that the rate of code flow begins to slow more and more, producing less than optimal integration results around major product milestones.

A closely related issue here is integration testing; in particular, the early detection and validation of key product integration points. If an integration test suite is not effective and efficient at validating the code being integrated as early as possible, then this can seriously exacerbate code flow problems. Our existing suite had many issues including a lack of: clear test selection criteria, formalized test case review/update processes, API-based automation, standardized logging mechanisms and test portability. Test collateral was too feature-centric, difficult to execute and debug for anyone except test owners, and often unstable. Years of growth had left the test suite with numerous ‘stale’ or redundant tests. The irony was that our integration suite was frequently executed and would pass without failures, yet product bugs, especially integration issues continued to be propagated during FI’s.

Finally, the issue of code flow and integration testing takes on a bigger dimension, when you consider the challenges of integrating products across Microsoft. Our partners, such as Windows, rely on us to provide them with high quality drops for their respective code bases. This involves defining a larger integration suite that is capable of expressing overall product quality respective Visual Studio to such internal Microsoft partner organizations.

In the remainder of this paper, we discuss our ongoing test improvements as part of the divisional “fearless FI” initiative. Section 2 focuses on the definition, implementation and execution aspects of the new integration test suite. In Section 3, we intend to deploy that new test suite to improve our existing code flow process within the division. For Section 4, we summarize the supporting test infrastructure and tooling needs that arose during development and deployment of this new suite. Finally, Section 5 hints at some of the future work that we intend to conduct in subsequent phases of this continuous improvement initiative.

2. Developing the New Integration Test Suite

2.1 Test Selection Criteria, Review and Resolution

The key objective for our test selection process was striking the correct balance between *efficiency* and *effectiveness* of the test suite. From extensive reviews of other integration test processes within Microsoft, it appeared that very few teams, if any, were achieving such a balance. Lack of clear selection criteria and more importantly, well-defined update processes resulted in integration test suites becoming bloated over time and significantly raising test costs over several product releases.

Based on these experiences, we concluded that it would be better to apply more stringent test selection criteria from the onset, possibly resulting in a minimal test suite. We would then leverage a well-defined update process to evolve the test suite alongside the product changes during the shipping cycle. Simple and clear criteria proved effective during communications with the teams. The following key criteria were set regarding *efficiency*:

- Tests execute in a two hour timeframe (excl. setup)
- Tests are reliable, consistent and easy to debug

We applied the following key criteria regarding *effectiveness*:

- Tests exercise key product integration points
- Tests reflect customer success scenarios (‘happy paths’)

As suitable test cases started to be identified, we categorized the resulting tests based on the number and uniqueness of the product integration points being exercised. The result was a primary and secondary set of integration tests, collectively known as the Reverse Integration Tests (RITs). These suites contained mostly functional, but also a few non-functional, namely performance tests; the latter needed to meet the above stated efficiency criteria.

The primary test set, or (Developer) Division RITs (DDRITs), focused on validating the major integration points that coincided with the top customer success scenarios. If these integration points were broken by any RI’ing team, then the product itself would also effectively be broken upon integration. This set of tests needed to be run as part of every RI and if any test from this set failed, it would block the integration of that team’s code into the main build. Any resulting bugs would be flagged as Priority 0 (highest priority) and need to be resolved within a few hours. This set of tests is currently of the order of tens of test cases to minimize execution and failure analysis time.

The secondary set of tests, or Product Unit-specific RITs (PURITs), focused on validating more team-specific, unique integration points that aligned with the remaining customer success scenarios. This set of tests would only be run by the team whom other teams took unique dependencies on. If any integration points were broken by the RI’ing team, then only the dependent teams would be affected; not the whole division. General code velocity would be maintained. Thus, any tests failing from this suite do not block integration for that team’s code.

Having said that, it is in the interest of dependent/partner teams to resolve the filed bugs (Priority 1) in a timely manner - any unresolved bugs and subsequent RI of code could lead to the dependent teams taking breaking code changes via an FI, which is to be avoided. In practice, teams selected those tests

that resulted in frequent breaking changes when their dependent teams churned code, that is, public APIs. This set of tests represents tens of test cases and will be run by the various teams across product branches.

The next step after identifying suitable tests involved the review and documentation of those proposed tests. In the case of the DDRITs, a *divisional* review committee was established comprising of both development and test managers as well as senior technical staff. Their role was to review and approve proposed tests and ensure strict adherence to the given selection criteria. While test case status was tracked manually, future support is being implemented using our work item tracking in the Team Foundation Server (TFS) product.

For the PURITs, the review committee comprised of staff from the partnering, dependent teams only. For those reviews, test purpose and content were captured via Service Level Agreements or SLAs between dependent teams. Templates were provided to the teams to ensure a general level of content consistency and then stored in a central Sharepoint server location to enable easy access for viewing and updating during the shipping cycle. Pertinent information included contact information such as test owners and delegates, list of all test consumers, anticipated test execution timeframes, links to test case descriptions and IDs, etc. Going forward, these divisional and team-specific review committees will become an integral part of our new integration testing process and are critical in helping teams define new RITs, update existing ones and deprecating old tests, whenever applicable. It will support our goals of maintaining an efficient and effective test suite.

While the above test sets may be sufficient for ensuring high-quality code flow within Developer division, they are insufficient when considering integration testing *in-the-large* across divisions. With this scenario, large portions of our code base need to be integrated into another division's product on a frequent cadence and have to be of very high quality. In response to these requirements, the test selection criteria were expanded to encompass the above types of integration tests as well as additional feature (functional), performance and stress tests that thoroughly exercise those code portions being integrated elsewhere. We also relaxed the execution constraints due to the larger size of this suite; enabling a runtime of about eight hours excluding setup and installation. However, we did maintain the test case reliability and debug goals – tests from either suite need to be stable and easy to debug! Unlike the RITs, which would typically be run against a product branch whenever a team wanted to RI (aka on demand), this suite would be run against those portions of code to be integrated elsewhere on a daily basis.

Given the different test suites and number of potential issues resulting from test failures, we aim to introduce the role of *Product Unit (PU) RI* representative as part of the process. In essence, each team will have such a contact. This person will be the first line of defense, should any integration tests fail; whether for product or test reasons. That person will be responsible for initial triaging and assignment of resulting bugs, collaborating and liaising with dependent partner teams to resolve, for example, any RIT issues, updating of SLA agreements, etc. From extensive reviews of other integration test processes within Microsoft, it appeared that those divisions that introduced such positions minimized the randomizing impact of incoming bugs on the teams, that is, developers and testers. We anticipate that these contacts will also be able to liaise with test infrastructure owners, if the bugs are tooling-related issues.

2.2 Test Implementation and Purification

With appropriate initial sets of RITs reviewed and approved, the next stage of our initiative focused on effectively implementing and 'purifying' those tests. We wanted to avoid the problems we had with our

previous integration suite and meet the criteria stated above concerning test execution *efficiency* – minimizing execution time, maximizing reliability, ease of failure analysis and debugging.

The vast majority of scenario-based test cases developed at Microsoft leverage UI-based test automation. This overexposure to a 'top-down' black-box testing approach has been beset by issues associated with performance, stability and reliability. In order to avoid such issues from the onset, we mandated that the primary set of integration tests, the DDRITs, must be implemented as *API-based* scenario tests. A new API-based test automation framework was developed and deployed to support this effort. Performance improvements were significant, often reducing execution time from several minutes to seconds; if the Visual Studio start-up time is not considered. It is worth noting that as tests utilized this framework, we also needed to make product changes for access to the appropriate APIs and improve testability.

In certain cases, however, we had to fall back on tests being implemented using a 'mixed mode' – containing both API- and UI-based automation, whenever the cost of refactoring product APIs to make them testable was prohibitive. With regards to the larger number of PURITs, we could only recommend that any new tests be implemented using API-based automation; essentially we leveraged existing test collateral due to time constraints.

The other important aspect of this work related to standardizing the logging data being generated by tests to ease the failure analysis for developers and testers as well as aid debugging by any team that breaks a RIT. For this, we defined a new logging API that is being integrated into our underlying API-based test framework and could easily be shimmed into an existing test environment for any PURITs. In future, we will be leveraging and extending this API as we explore smarter failure analysis techniques.

Once implemented, all test cases were then subject to 'purification'. This meant that tests were repeatedly executed against a predetermined matrix of OS, Visual Studio SKU configurations and languages. In each case, the test execution system would monitor whether test failures occurred during execution. Tests failing purification could be repeatedly submitted until they ran flawlessly. Future versions of the purification system and process will be implementing 'test pollution' reduction measures that ensure tests leave their execution environment pristine. In future, we aim to also develop static analysis checks that enforce test design practices for UI- and API-based test development. These automated checks would run prior to test execution.

2.3 Test Execution and Reporting

Test execution and reporting leveraged the results of two related, divisional test initiatives. The first effort focused on consolidating our existing, distributed testing lab resources by moving them into a new cloud infrastructure at a geographically remote site. There, a large number of virtual machines (VMs) became available, which teams could use for testing purposes based on virtual machine images (VHDs). In addition, this effort included enhancements to our existing test case management system to enable execution of the functional RITs in the cloud.

In the past, a large percentage of time, prior to test execution, was spent with basic machine configuration tasks including the setup and installation of OS/SKU/language configurations. With the advent of clouds, such setup and installation work is minimized. A priori, VHDs are defined containing all preconfigured OS/SKU/languages configurations necessary for the test runs. These VHDs are stored as part of the cloud infrastructure. Each test run can then quickly image the VMs, install the required Visual Studio and related products and begin test execution.

Upon test failure, whether product- or test-related, the VM executing the erroneous test is saved for the developer or tester who can then remotely debug the issue. Once the issue is resolved, the VM is released back to the pool of machines.

The second effort being leveraged for integration test purposes is focusing on consolidating our heterogeneous and distributed reporting mechanisms via a central 'dashboard'. To increase transparency, that is, provide quicker and easier access to detailed integration testing results, a new reporting infrastructure is being developed that will enable the entire management chain to consume and respond to this data. This is especially important in the case of the RITs, whose execution results need to be quickly scrutinized for failure in order to respond in a short time period.

In summary, the new integration test suite can be executed by our central engineering team (DES) on behalf of our RI'ing product teams with results being reported and available for all to view in one location.

3. Deploying the New Integration Test Suite

The above sections described how we defined, implemented and executed the integration test suite. We also introduced the different stakeholders and their roles/responsibilities - central engineering team (DES), divisional and team-specific review committees, PU RI Rep and PU Developer and Test Owner. In this section, we describe how we intend to deploy the new test suite as part of our improved integration test process.

3.1 Deployment Goals

We set ourselves the following deployment goals:

- **Maintain test suite efficiency** – the performance and reliability of the tests needs to be maintained under all circumstances; maintain rapid code flow is paramount.

- **Evaluate fault detection capability** – the effectiveness of the new integration test suite should be at least on par with the old one; the most important integration bugs should be caught by both suites.

- **Evaluate test suite effectiveness and adequacy** – the new integration test suite should continue to exercise key integration points in the product as well as reflect customer success scenarios as the product evolves, that is, code churns.

3.2 Workflow

The workflow and process is currently being defined and is subject to change. Having said that we have defined the two key scenarios that are being fleshed out:

a) A team is ready to integrate or RI their code contributions
b) Product changes require the test suite to be updated

3.2.1 Preparing for Reverse Integration

Figure 1 depicts the workflow throughout reverse integration of a given team's code contributions. With the latest daily product branch (PU) build available from the build team, the team wanting to reverse integrate their latest code changes requests an integration test run from DES Test, the central engineering test team. They, in turn, schedule a run of all applicable integration tests in the cloud. If all tests pass successfully, that is, the reporting dashboard lights up green, that team's code contributions can be submitted and merged into the main product code base. Consuming teams should in theory be able to take a fearless FI and not be broken.

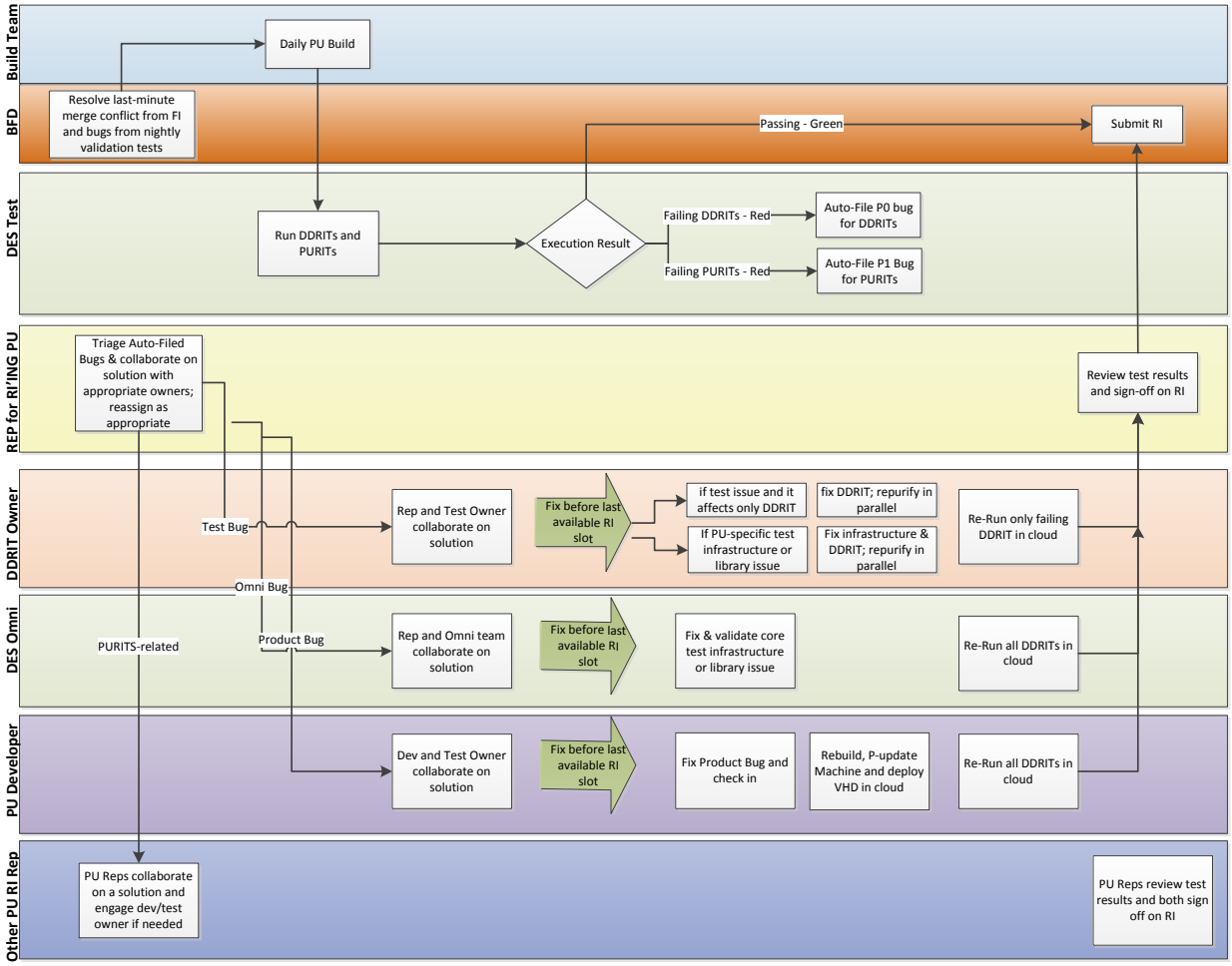


Figure 1: Integration Testing Process during RI

Otherwise, the reporting dashboard would light up red. Bugs are automatically filed with status appropriate to the test case – P0 bugs for DDRITS and P1 bugs for PURITS. The representative for the RI'ing PU conducts the initial triage of the incoming, auto-filed bugs for DDRITs and reassigns them as follows:

- **Product bugs** are filed against developers
- **Test bugs** are filed against test owners
- **Test infrastructure bugs** are filed against Omni team (centralized engineering team)

The representative also initiates discussions with other product teams, if the failure is PURITs-related. With these test failures, the workflow has been prescribed and agreed upon in the service level agreement (SLA). It follows a pattern similar to the one described in the next few paragraphs. If bugs are resolved in an appropriate and timely manner, the representative will review and close any outstanding bugs as well as sign-off on the RI.

For developers, this process implies that product fixes are necessary to the PU build. Once completed, the *entire suite of applicable RIT tests* needs to be rerun to ensure no regressions. To achieve this, the PU build must be updated and a new VHD created for rerun in the cloud. Developer and PU RI rep must sign off on the work upon completion.

For test owners, this process requires fixing of the test bug and revalidating the *affected RIT test case* to ensure no regressions. Re-purification is also necessary to maintain test suite efficiency, but does not block sign-off. Test owner and PU RI rep must sign off on the work upon completion.

For the DES Omni team, this process involves fixes to our core test frameworks and/or libraries. As these frameworks impact all RITs tests and are critical to the success of every team's RI, this central engineering developer team needs to rerun the *entire suite of applicable RIT tests* to ensure no regressions.

Up to 5 RI slots are available per day for teams to RI their code. With such an RI schedule, it is possible for teams to spend the best part of a day to resolve their P0 and potentially their P1 issues resulting from failed RITs. If DDRITS-related bugs cannot be fixed within that timeframe, main product builds will be reverted to previous day's state.

Reporting is an important cornerstone of monitoring adherence to the original selection criteria as we enter the new shipping cycle. It will also act as the trigger mechanism for updating the existing test suite, if necessary. The central engineering team will be providing reports that examine the health of the integration suite by trending on the following types of data:

- **DDRIT /PURIT passing rates** (identifying who is to be in "test result hell")
- **Code velocity cadence** (identifying outliers, which teams are slow and fast to RI/FI)
- **Product and test gaps** (types of failures, customer scenarios or integration points being missed by which test suites)

3.2.2 Updating the Integration Test Suite

Given the above workflow and trending data/reports, we are currently addressing the challenge of updating and evolving the new integration test suite, once the product churns. During such churn, when significant feature enhancements or new customer scenarios are added, feature code as well as their integration points would be affected. The test suite would need to be adapted. Two scenarios are possible:

- **Integration test(s) fail during RI** – in this situation, *previously traversed* product features and integration points have been significantly modified or deleted in the RI'ing product branch to the

extent that one or more tests fail. As a result, the PU test owner needs to significantly rewrite one or more tests. For the DDRITs, this would result in new reviews of the updated tests by the divisional committee who, upon approval, would request renewed purification of that test case. For the PURITs, two PU RI reps need to decide whether the tests need to re-reviewed and/or re-purified. Details should be specified in the team's service level agreement.

- **Integration tests pass, but breaking changes occur upon FI** – in this situation, teams taking fearless FIs experienced failures due to significant modifications or deletions to existing product features and integration points. These features or integration points were *not previously traversed* by integration tests and given the severity of breaking changes, it may be decided that new integration tests need to be implemented or old ones modified/enhanced.

4. Supporting Tools and Infrastructure Work

We realized early on that significant changes were needed to our existing test infrastructure as well as additional tool support to support the definition, implementation, execution and most importantly updating of this integration test suite. In this section, we wanted to highlight three key improvements that are currently under way.

4.1 New Static Analysis and Dynamic Testing Tools

We are developing a set of static analysis and code coverage-based tools that enables us to capture and visualize function-level dependencies between product branches. We can then overlay those views with code coverage and code churn data. This enabled us early on in the initiative to support teams with identifying dependencies during the test definition phase and will in future enable us to validate the coverage of key integration points as part of test execution. Code churn data will be used to address the update issue by highlighting public APIs that have changed and thus integration points that may have been affected. For example, by analyzing the dependencies between teams A and B, we may identify that no dependencies exist or have been modified during the RI of team A's code, so team B can take their FI with a high degree of confidence that they will not be broken by team A. Alternatively, the tools may indicate that team B does have dependencies on team A's RI'ing code, but coverage data indicates that no RIT coverage is available. Team B can plan for some additional testing, suggest a new PURIT test to team A, if the dependency is unique or propose a new DDRIT to the division, if that dependency is taken by most teams.

4.2 API-based Test Automation Framework

Having learned from past and often painful experiences, our existing UI-based test automation needed to be complemented with an effective and efficient API-based approach for developing these integration tests, in particular, the DDRITs. The challenge was also one of consolidation as over the years, a plethora of API-based automation frameworks had emerged and evolved. The resulting framework helped us standardize around one tool, provided a uniform access mechanism to the Visual Studio Object Model (OM) and implemented standardized, yet extensible test code logging mechanisms. With access to the OM, the resulting tests cases demonstrate significant performance gains over the UI-based tests making the DDRITs tests fast and effective.

4.3 Test Portability

In the past, our existing integration test suite suffered from test portability-related issues. Teams were not necessarily able to run each other's tests early on as part of the RI process and if they did, it required considerable resources to execute and debug those tests. However, one of the requirements for the new integration test suite was that tests should be portable meaning that they can be *run by anyone*. This requirement has led to RITs test owners clearly specifying as well as reviewing and rationalizing the execution needs, dependencies, etc. for each test. As a result, integration tests can be run in the cloud by our central engineering team or any staff member across the division without additional effort.

5. Conclusion and Future Work

In this paper, we described an ongoing test improvement initiative whose goal is to help reinvigorate our code flow, both in terms of velocity and quality. We examined the key challenges facing us with large-scale integration testing at Microsoft and described our response in terms of the development of a new integration test suite. Details were given concerning the definition, implementation and execution of this new test suite as well as an overview of the anticipated workflow that leverages the test suite. Moreover, we outlined the tools and infrastructure changes that were required to support this effort.

At the time of writing, we have started to execute our new integration test suite regularly against the existing product code base and will be validating the results reported against our stated deployment goals. Based on those results, we will be focusing on developing more supporting tools and infrastructure that can help us evolve the existing tests.

6. Acknowledgements

I would like to thank Larry Sullivan, Director of Engineering and David Sauntry, Principal Architect Lead, for their ongoing support. I also want to express my gratitude to Steve Arnold, Richard Kuhn and Mark Osborne for their support and discussions regarding the issues discussed in this paper. Thanks also go to my reviewers whose invaluable feedback and comments are greatly appreciated.

User Experience Grading via Kano Categories

Matt C. Primrose
Visual Computing Group
Intel Corporation
Portland, OR. USA
matt.c.primrose@intel.com

***Abstract*— A consistent challenge in product development is determining whether a product meets its usability requirements. In particular, accurate, meaningful usability information is hard to obtain before product release. This paper describes a method to classify use cases, features, and requirements into Kano model categories and then grade each based on how well is has been implemented from a usability perspective. Using this method during product development can provide early, regular data on the status of product usability, help to determine where resources are spent, and perform competitive analysis based on product features and usages. This method was tested on a single case study, and in particular for the requirements of the user interface for a discrete video card.**

User Experience; Kano; Usability; Requirements Verification;

I. INTRODUCTION

Usability is by nature a subjective measurement. Each person may find a product more or less usable than others based on his or her own perceptions and experience. The best that developers can hope for is to create a design that the majority of users will find easy to use while still meeting all of the functional requirements.

The method this paper describes is not a panacea for the problem of measuring usability but instead can be used by development, validation, and product marketing to help guide developers in making good choices when it comes to allocating resources, cutting or keeping features, and whether to add that "one generation ahead" feature or leave it out until the next version.

The process flow [Fig. 1] outlines how this method works. The shaded boxes in the flow are where this method adds value to the standard planning and execution flow and will be the focus of this paper. Much of this process work happens during the product planning phase. There is also a small grading effort and usability assessment required during development.

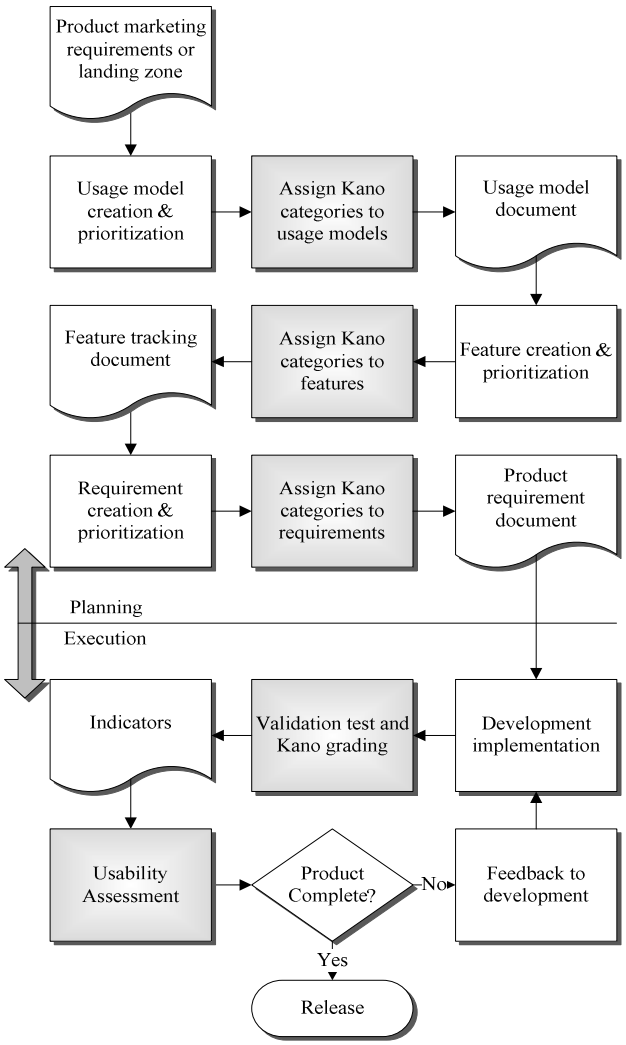


Figure 1: Process flow

II. KANO MODEL

The method of grading and prioritizing development work described here is based on the categorization of use cases, features, and requirements using the Kano model [1]. The Kano Model of Customer (Consumer) Satisfaction classifies product attributes based on how they are perceived by customers and their effect on customer satisfaction.

These classifications are useful for guiding design decisions in that they "indicate when good is good enough, and when more is better" [2]. Following the principals of a simplified version of the Kano model [Fig. 2], each usage model, feature, and requirement is categorized into one of three categories: Must Have, Desired, or Differentiator.

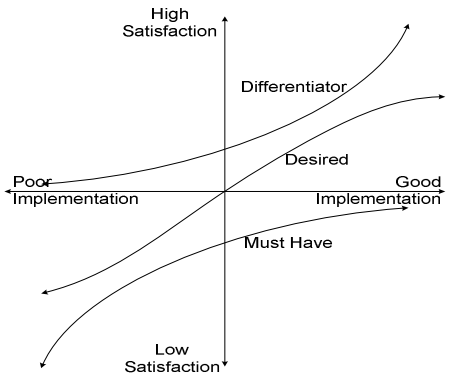


Figure 2: Kano graph

The "Must Have" category includes those components that customers expect to be available for the product being designed. A product missing one or more "Must Have" usages, features, or requirements can result in customer dissatisfaction and leave customers thinking that the developer did not understand the needs of their consumers.

"Desired" usages, features and requirements are related to performance measurements. These could include how fast a product responds to user input or how much of a usage, feature, or requirement is available to the user. The more that the user experience exceeds the minimum threshold the more customer satisfaction will be generated by the product - until a point of diminishing returns is reached. The threshold value as well as the diminishing returns value should be accounted for in the product requirements to ensure that the right amount of development effort is spent.

"Differentiator" capabilities are what uniquely differentiate one product from other similar products. "Differentiator" usages, features, or requirements tend to be transitory in nature. Once consumers have accepted these capabilities, they quickly come to desire and expect them to be available in future products. Therefore, a new usage, feature, or requirement classified in the "Differentiator" category can expect to be there for one generation of the product and then, as customers desire this capability and come to rely on it, may be re-categorized as "Desired" and then "Must Have".

III. THE PLANNING STAGE

Product development can begin with a business opportunity, compelling new technology, innovative usages, or any combination of the three. In order for a product to achieve broad success, however, it needs to balance all three of these perspectives [3]. To this end, product planning must include business analysis, new technology architecture, and a description of the intended usages for the product. These intended usages are at the core of what this usability grading method uses to provide feedback to developers.

Based on the fundamental business, technology, and usage data, the product features and requirements are written and prioritized so that developers know exactly what the final product must (and in some cases, must not) do. Once features and requirements are specified, those that affect the usage models are categorized into Kano groups in order to drive the usability assessment.

A. *Applying the Process*

Categorizing the usages, features, and requirements into Kano categories takes roughly the same amount of time as it takes to assign traditional priority. The advantage of using Kano categorization instead of prioritization is that simple prioritization is one dimensional whereas Kano categorization is able to provide important information in addition to priority and motivation at the usage, feature, and requirement level.

Unless a requirement, feature, or usage is removed from the product, all requirements should be implemented before a product can be considered feature complete. By using Kano categories instead of simple prioritization, developers can implement requirements based on related features and how those requirements affect the feature. This enables a logical development flow that groups requirements together based on feature, and prioritizes the requirement implementation. Requirements critical to consumer acceptance of a feature are implemented first, with enhancements to those features being done only after the minimum levels of achievement that define success have been reached.

To further explain how this development prioritization process works, consider the example of a product that adds value to three usage models. The first usage is categorized as a "Must Have", the second usage is classified as a "Desired", and the third is considered an "Differentiator". Each of these usages will have associated features that have their own Kano categorization. For the "Must Have" usage [Fig. 3], each feature could be a "Must Have", "Desired", or "Differentiator" feature depending on how that feature relates back to the usage. In addition, for each feature, the requirements supporting the feature will have their own Kano categorizations which may not necessarily match up to the feature categorization but instead may be based on how the requirement relates to the feature. The priority by which each requirement is implemented is based on the criticality of the requirement to the feature and then the feature to the usage.

Generally, the "Must Have" category has the highest priority. Requirements in this category provide the base level of functionality for the features and usages and in most cases should be developed first in order to establish the product framework. Requirements in the "Desired"

category have the next highest priority until they meet their minimum threshold. Once that threshold has been met, the priority for additional development for the "Desired" requirements lessens until the maximum threshold (point of diminishing returns) is reached, at which point the priority is reduced to nothing. "Differentiator" requirements tend to start out with a lower priority compared to "Desired" requirements, but once the "Desired" requirements have reached the minimum threshold, the priority for developing "Differentiator" requirements increases and surpasses the priority for additional development of "Desired" requirements. Time permitting, additional development of "Desired" requirements should happen until the maximum threshold is reached. The example [Fig. 3], shows a number in the requirement box which represents the order in which the requirements would be implemented. Exceptions can be made due to cost associated with implementation, technology readiness, or a staged release process, but in general this is the way the category priorities work best.

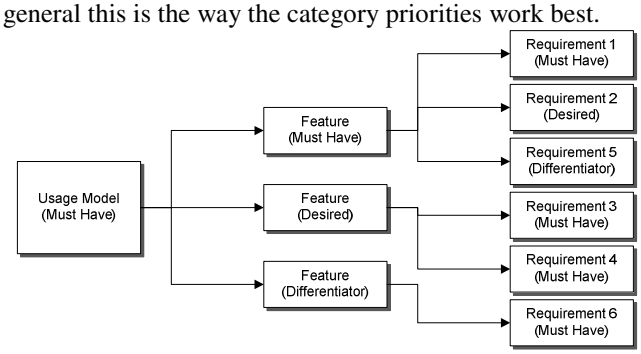


Figure 3: Categorization tiers

Once product development has moved into the development phase, developers can use the Kano-categorized usages, features, and requirements to prioritize the development and validation work. Knowing which requirements relate to which feature provides the ability to assign similar work to the same people, increasing development efficiency. Well-defined requirements aid this efficiency by describing what needs to be implemented and, in the case of "Desired" requirements, the minimum and maximum amount of implementation required.

In some cases, features may relate back to multiple usage models, or requirements may relate back to multiple features. In these cases the Kano category for how each requirement as it relates to each feature and each feature as it relates to each usage model needs to be tracked. Having this information available at the feature and requirement level allows for better decision making, resource allocation, and prioritization. If requirements are altered or removed during the planning or execution stages, a clear impact assessment can be made for each feature and usage related to the requirement being changed.

B. Planning Validation

Kano categorization during the planning stage is not just preparation work for the execution stage of development. Comparing the product's planned usages and features to competitive products can be done to determine if the current usages and features described will deliver a compelling product. By performing a feature analysis using Kano categories of a current product and comparing that to the features of the planned product, a competitive analysis between the two can be performed. Taking into consideration that competing products are likely to be enhanced during the product development period, features, requirements, and usages can be modified to ensure that the product being developed does not just meet the current competition but can compete with potential products that will be available at launch.

IV. THE DEVELOPMENT STAGE

Validating that the product being developed meets requirements is an important part of any product development. In most cases this validation work happens in conjunction with development work to minimize time spent at the end of the project fixing bugs or improving usability. Kano grading fits in well with this form of testing by forcing validation teams to look at each requirement, feature, and usage to verify both *if* it is implemented and *how well* it is implemented. As an added benefit this level of examination quickly exposes any unclear requirements and provides opportunity to clarify confusion between the development, validation, and marketing teams.

Generally, developers implement requirements for a given feature at the same time. These requirements are usually related to one another and verifying that they work together to enable the feature is easier when they are developed simultaneously. As features are enabled in a product, the validation team needs to test to verify that the feature as a whole is meeting expectations. While requirement testing is more geared for verifying that each individual requirement is implemented correctly, testing at the feature level will mostly find functional and usability issues and should be designed to test for these possible issues. Assessment at the feature level is a great opportunity to find these issues and make adjustments before the product moves too far down the path of development.

While most testing is done at the requirement and feature level, assessing overall product usability must be done at the usage level. The usages are what customers of the product most relate to and usages provide the root-level rationale for the user-based features and requirements. If the requirements for the product are written and implemented correctly and any usability issues at the feature level have been resolved, performing the usage-based usability assessment should find few new major issues: issues found at this level mostly center around interoperability between features and minor tweaks to overall usability of the product. If major usability issues are discovered at this level, then either something was likely missed in the planning phase or the market changed significantly in a way

that renders the current product implementation at least partially obsolete.

Usage grading generally lags behind feature assessment much as that feature assessment lags behind requirement testing. As features and usage models are implemented, validation is able to ramp up the testing in these areas and reduce the overall amount of requirement testing [Fig. 4]. As requirements for a given feature are enabled, feature testing can ramp up and replace many of the requirement tests that were previously done. The same transition occurs for feature tests being replaced by usage model tests. Because of these factors, as the project progresses the focus of the testing is more complex and closer to what end users will experience. Over time, an increasingly accurate assessment of usability is gained. However, a base level of requirement and feature testing still needs to be maintained to catch regressions.

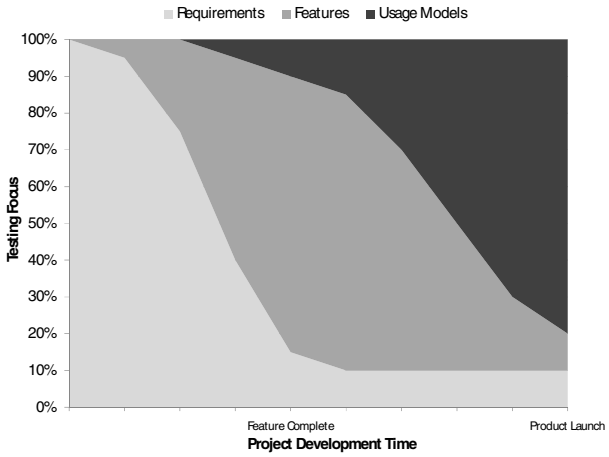


Figure 4: Validation testing mix

A. *Validation Grading*

Validating whether a usage, feature, or requirement is implemented is fairly straightforward with a standard pass/fail result; however, to measure *how well* a usage, feature, or requirement is implemented requires a more robust method. A grading scale is able to handle this in a way that allows for quantification of a subjective assessment. The grading scale this method uses categorizes the usages, features, or requirements into four categories: Not Implemented, Partially Implemented, Fully Implemented, or Implemented Beyond Minimum Requirement. Each category is assigned a numerical value [Fig. 5] which is then used to generate the reporting indicators.

Assessment Result	Point Value
Not Implemented	0.0 points
Partially Implemented	0.5 point
Fully Implemented	1.0 point
Implemented Beyond Minimum Requirement	1.25 points

Figure 5: Kano grading

Usages, features, and requirements not yet implemented score a zero. Partially implemented items score a half point. A "Partially Implemented" assessment may be due to the requirement, feature, or usage being dependent on other requirements, features, and usages. Those requirements, features, and usages that are fully met are given one full point. Items that exceed the minimum, for example a product that performs faster than the minimum customer requirement, are scored at 1.25 points.

Each usage, feature, or requirement is graded separately. Each must be fully implemented or have an approved exception in place before the project is complete. For example, four requirements that are scored at 1.25 do not make up for a single requirement scored at 0. The scores help to evaluate how the project is progressing, identify what areas of the project require more development, and aid in assessing the usability of the product in order to meet project goals.

Generating test results for Kano categorized requirements is different from a standard pass/fail test. For results that would normally be a "fail", tests must be differentiated between "Not Implemented" and "Partially Implemented" results. And "passing" results must be refined into "Fully Implemented" and "Implemented Beyond Minimum Requirement". For example, a test for assessing the requirement "The user interface must support a minimum of 7 profiles and a maximum of 35 profiles" could have one of four possible results. The "Not Implemented" result would be if the user interface did not have profile support at all. The test would be marked as "Partially Implemented" if the user interface had profile support but either the profile support was not currently working (placeholder) or it supported less than the minimum of 7 profiles. If the testing showed that the user interface supported 7 working profiles then a result of "Fully Implemented" would be recorded. And finally, if the user interface supported more than 7 working profiles, then a result of "Implemented Beyond Minimum Requirement" would be given. If more than 35 profiles were supported, the assessment would stay the same; however, customer satisfaction would not be significantly improved by going beyond 35. Each result is then converted to its numerical equivalent [Fig. 5] and can then be used to generate testing indicators.

While performing the grading it may be that "fully implemented" or "partially implemented" scores decrease from one test run to another. Usually this is a sign that a regression has occurred in the product and should be treated just as if any other test had failed. An "Implemented Beyond

Minimum Requirements" assessment result that drops is not cause for a failure report (unless it drops below "Fully Implemented"), however it may indicate that a performance issue is occurring and could warrant additional investigation. As with other forms of testing, the tests used in these assessments must be kept aligned with the current requirements, features, and usages - otherwise false results could be recorded, or defects could be overlooked.

B. Assessing Usability

Usability assessment is a mix of subjective and objective measures. Additional changes can always be made to satisfy some users, but those same changes could negatively affect other users. The usability goal for designing any product is to make the product the most usable for the target users while providing the desired feature set.

To help achieve this goal, the scores are used to assess how well the currently planned requirements, features, and usages are implemented. If the product planners did a good job of identifying the important usages and the requirements were well written, the scores should provide an accurate baseline for the product usability. Increasing scores indicate a more complete and more usable product. For the "Must Have" category, the highest possible score is 1.0 and the average score for all "Must Have" usages, features, and requirements must be at 1.0 before a product launches. Exceptions are allowable with the understanding that if there are "Must Have" requirements, features, and usages that are not scored at 1.0, then customer satisfaction will be greatly affected. For "Desired" and "Differentiator" categories, the maximum score is 1.25. In order to ensure good usability for users, the average validation score needs to be between 1.0 and 1.25 across all requirements, features, and usages. If one of these items is not scored at 1.0 or higher in time for product launch, then customer satisfaction for that feature or usage will be negatively affected and product adoption could be slowed or narrowed. Missing a grade of 1.0 for any requirement, feature, or usage should be considered a test failure and appropriate bug reports filed.

In addition to validation grading, competitive analysis can also be performed using the same Kano category grading technique. The grading of a competitor's product is easier to perform at the feature or usage level as access to their development requirements is typically unknown. By comparing the grading results of the current competition product and the current in-development product, a competitive analysis can be made. Of course, competition does not stand still waiting for other products to be developed, so it is good to perform this competitive analysis every time a competitor releases a new version of their product to keep abreast of new features and usages.

C. Feedback to Development

The results of the validation testing need to be rolled up to the development team to make them aware of any issues found in the testing. As issues are found and fixed, tests must be re-run to verify that the fixes were implemented properly and new issues are not created. Results from competitive analysis can cause shifts in the product priorities. Working together with marketing and development teams, these shifts must be managed to keep the product relevant to customers without significant impact to development schedules.

As a product goes through validation cycles opportunities can arise to root-cause usability issues that were unforeseen when the product requirements, features, and usage models were initially developed. This is where qualitative measures potentially will have to be used in order to identify if the usability issue being seen is something that should be changed or is serving the intended purpose. It can be difficult to fully comprehend how a design will look and feel without being able to interact with it. Leaving open the possibility for small design changes during implementation allows the developer, marketing, and validation people the ability to adapt to a changing market.

V. CASE STUDY

The User Experience Grading process was developed and implemented by a validation team working closely with software development and product marketing. The project was a control panel user interface for a discrete video card. The development team used the Scrum method of Agile development, which worked well with the User Experience Grading process described in this paper. The sprint cycles allowed for regular grading checks and feedback throughout the development process. Due to the timing of the project and the point in which the validation team was engaged, the User Experience Grading process was only used at the requirement level (features and usages were not graded), and only after working code was available to test. Even with these limitations to the process, the value of using the User Experience Grading process over traditional validation methods paid dividends by catching and resolving issues earlier in the development process, forced better requirements management, and reduced development team overhead.

Overall, the case study included 266 requirements graded over 11 sprints (10 months). To date, the User Experience Grading process has helped find over 400 functionality and usability issues with the control panel software. The product this process was used on was an internal-only version of the video card control panel; however, many of the usability issues found on the internal version of the software were used to improve the external version of the same software design which received positive reviews by industry press. One review stated "*[Company] has also been working to improve the user interface of the graphics control panel. What we saw looked much improved over the existing [company] control panels.…"* The ability for validation to engage earlier and provide feedback to the developers in a way that highlighted usability enabled solutions to be developed that could then be used on similar products being developed in parallel.

VI. BENEFITS

Despite applying the User Experience Grading process only to requirements, there were several observed benefits.

Early Detection: Making use of the User Experience Grading process the validation team was able to identify several issues during the pre-alpha phase of software development that could have potentially been missed until much later in the validation cycle. Using the process required the validation team to look at each requirement and grade how well the current functionality compared to the wording of the requirement at the same time subjectively assessing usability for every sprint cycle. Performing the checks as individual requirements were implemented in code by the development team (even before development had fully implemented a feature) enabled early detection and correction of potential issues.

For example, the user interface aspect of a new feature was introduced during a sprint cycle without the underlying hooks into the driver. Validation used the associated testing cycle to assess this new code against the documented requirements and for usability. Validation found logical flow issues that could have turned into significant usability issues. This early identification allowed the development team to make a relatively easy change to the code and resolve the issue before the impact of such a change could become costly.

Better Requirements Management: Another benefit found during the case study was the ability to find and adjust requirements whose wording needed to be updated due to changes in project scope. By regularly grading the code against the requirements and reviewing the grading with development, the team was able to quickly find places where an implementation decision had been made but the requirement had not been updated to match. This constant checking of the code against the requirements naturally created a forcing function to keep the requirements up to date. Of course, when a discrepancy between requirements and implementation was not due to an approved change, it resulted in a defect report rather than a requirements update.

Reduced Development Team Overhead: Before the validation team began using the User Experience Grading process, the development team had kept its own set of the tasks and features they were working on at any given time. Using this list, they determined what parts of the code were complete and what parts were not complete. This work was overhead that the development team had to track on a sprint by sprint basis and realistically should be done by the validation team. By using the User Experience Grading process, the validation team reduced the overhead for the development team while at the same time increasing the quality of the validation feedback.

VII. LESSONS LEARNED

Future projects: It is suggested to use the User Experience Grading process earlier than was possible in the case study. Making use of the Kano categories and competitive analysis in the planning phase would have helped to determine better prioritization and focus on usages and features most important to the target users of the case study product.

Using only the requirements for grading limited the scope of the validation team's feedback to the development team. Expanding the grading to include both the features and the usages would enable validation to take a more holistic approach to evaluating the product's usability. In the case study, feature and usage level usability validation had to be done using "gut instinct" and expensive semi-annual user studies. Identifying and changing subtle usability issues was difficult without having user study feedback available. Having a validation process in place like the User Experience Grading process that is equipped to evaluate the usages, features, and requirements would have reduced the need to host as many user studies and would have enabled faster turn-around time for resolving usability issues.

REFERENCES

[1] N. Kano, et. al., "Attractive and Normal Quality", *Quality*, vol. 14, no. 2, 1984

[2] http://www.ucalgary.ca/~design/engg251/First%20Year%20Files/kano.pdf

[3] E. Simmons, "The Usage Model: Describing product Usage during Design and Development", *IEEE Requirements Engineering*, 2005

Affecting Printer Installation Success in the Consumer Market

Kathleen Naughton
kathleen.e.naughton@hp.com

Abstract

This paper will describe the design and development of a test lab that has been affecting change in the R&D requirements and development processes to reduce the install call rates while supporting the adoption of wireless technology advances in the consumer customer segment. The paper will:

- Describe the process of aligning on a definition for install success
- Identify factors that contribute to install success and failures
- Explore how unactionable, field-found issues influenced test lab design
- Describe how test lab design objectives drove executive sponsorship, development, and implementation of a whole new testing service
- Show how the whole process has driven changes into R&D requirements and deliverables
- Examine recent call-center data to validate affects of these changes

Biography

Kathleen Naughton is a software test architect for Hewlett-Packard. Kathleen has over 20 years experience in the technology industry implementing a parts ordering system on one of the first nation-wide WANs for Tektronix, verifying graphical software development kits for CA, designing testing for streaming media cell phone software with Packet Video, and most recently, printer and scanner test design and lead for Hewlett-Packard's Imaging and Printing Group. She has been the printer installer test lead for over 5 years, co-chaired the Install Success Taskforce, and directed multiple programs and studies performed to reduce the install call rates. She is an active member of the Anita Borg Institute's organizing committee for the Grace Hopper Celebration of Women in Computing conference as well as a regular presenter at that conference.

Introduction

I just bought a new printer—needing to get my [term paper, resume, wedding invitations, soccer team flier] printed like yesterday. I unbox the device, follow the single-sheet "how to" instructions and I'm good to go. But wait! My PC can't find the printer on my wireless network. My security software keeps giving me pop-up notifications and warnings. And now I have gotten a fatal error from the installer! On the third attempt and with assistance from the "dreaded" call-center technician, I am finally ready to print that essential document that initiated this adventure into technology setup and configuration frustration and aggravation.

This scenario is more common than we as technology companies want to admit. It is an especially challenging experience in the consumer and small business marketplace where our PC and network environments are not managed by IT department policies and processes. At HP, we have categorized call-center calls to help us monetize the cost of failures in the field. We knew the data – at one point 70-75% of calls were categorized as "install" calls. We developed tools to help us mine what the failures were. But we could not seem to effectively change the install call-center rates in the consumer marketplace. That has been changing!

This paper will describe the design and development of a test lab that has been affecting change in the R&D requirements and development processes to reduce the install call rates while supporting the adoption of wireless technology advances in the customer segment. The paper will:

- Describe the process of aligning on a definition for install success
- Identify factors that contribute to install success and failures
- Explore how unactionable, field-found issues influenced test lab design
- Describe how test lab design objectives drove executive sponsorship, development, and implementation of a whole new testing service
- Show how the whole process has driven changes into R&D requirements and deliverables
- Examine recent call-center data to validate affects of these changes

Aligning on definitions

It seems obvious what it means to have a printer successfully installed, but install success has as many meanings as there are perspectives of looking at it.

We use our call center data as a direct conduit to customer experiences. Our call center "install" bucket was filled with calls regarding network failures, printer hardware power issues, ink cartridge installation difficulties, unpacking confusions, paper loading location misunderstandings, failures to print or scan after software drivers were seemingly successfully installed, and errors that occurred as the software was installing. From the user's perspective, they are all install failures. The software installer development team became the "owner" of all these issues because they are the only team with "install" in their name.

As a group assigned to look at the call center install failure bucket began to dive into the call data, it was quickly realized that the team needed to be more than just the software installer team. We created a cross-functional team and began re-defining the call center data. Since the "install" bucket actually encompassed failures that occur in the "1st day of ownership", we called the large bucket the "1st day experience" issues. We then further divided the issues into software, hardware, product design, and "other" buckets. The team then divided back into their respective functional areas and began working on attempting to address the different types of issues. The remainder of this paper will look at the software 1st day failures and how—through a series of tools, requirements, and, customer test programs—a new testing service was designed.

With the narrowing of the view of install failures to the software portion, we still needed to align on what install success really means. By definition of existing release standards and criteria, we were having 100% install success through defined testing methods and lab execution programs. These methods and execution programs were defined by program requirements generated by our Marketing and Customer Satisfaction groups, and by technical requirements generated by the R&D community. Our lab methods and executions used the principle of isolation such that we used clean operating systems with nothing else installed on the system. We also used relatively fast hardware and larger memory to allow for higher throughput of tests. However, our customer install success rate was still not improving as measured by the call center data, so we needed to better define what install success meant to the software lab to help focus the requirements and prioritization.

We developed a multi-part install success definition:

- Technical Install success: there are no errors generated by OS or installer, and communication between printer and PC is verified
- Install time success: a specified time limit that installs are expected to be complete
- User experience success: the customer can successfully navigate through the required steps to complete the installation with a functioning printer

In-depth examination of the User experience success will not be addressed in the remainder of this paper; it is still an evolving challenge. The other two definitions are the core driving definitions for R&D driven requirements and the focus of a new testing service.

Contributing factors to success and failure

Now that there was alignment on what install success was, we needed to see if we could figure out what contributes to install success and especially to install failures. Since we were experiencing 100% install success in our lab test environments, this required some out-of-the-lab-box approaches. The backdrop for all this work was the continuous product release cycles that we needed to deliver. On average, the product release cycle had major software product releases occurring every six months. These product releases needed to not only support new printer device features, but also major technologic advances and operating system releases. The technologic advances include moving from primarily USB connected devices to network connections and then to wireless connected PCs and devices. The operating system releases included support for 64-bit architecture and major OS changes from Windows 98 through Windows 7.

One of the first things we needed to do was to figure out how to analyze the failures occurring at the customer locations. We needed a way to do this remotely and preferably at the time of failure. The development team created an error recovery tool that would generate a hash code based on an algorithm that included the last executed line of code in the installer and other factors that would help isolate the state of the installation at the time of failure. Then, with customer approval, we would collect this hash code and system configuration information into a package and post it to secure HP servers. This took some refining and evolved to also include a download to the customer's PC of any fixes for matching hash codes. The main problem with this tool is that even with this information, we could not consistently reproduce the errors in a lab setting. So the number of fixes that actually got developed and deployed was very limited. We were still releasing products with similar software install failures so the call rates remained statistically unchanged.

The next step in the analysis was to try to figure out how to be there when customers experienced failures. We had an existing customer Beta program that took our products with HP observers to customers' homes. The program entailed having customers take a boxed printer and go through the entire setup from unboxing, setting up the printer (installing cartridges and paper, plugging it in), installing the printer software and drivers, and then using the printer. The HP observer's task was to document what the customers were experiencing—any missteps they took in the process, any technical issues they may have experienced—as well as documenting the customer's environment—their PC configuration, operating system, networking, and any installed software. The program left the device with the customers

for 2-4 weeks to gather in-depth usability feedback. This was a good source of information. However, there were three major flaws to the program: first, it was too late in the program to allow for issue fixing unless we delayed the product release; second, the sampling pool was too small—usually about 15-20 customers; third, we had very few technical install failures in the program.

R&D needed the information earlier in the project to be able to make any code changes to correct any issues. They also needed a larger sampling size so that we could be able to generate statistical information. A failure in a sample size of 20 (5%) is significantly different from a failure in a sample size of 50 or 100 (2% or 1% respectively). So the Iota program was developed. Essentially, it is an install-only beta program. We leveraged the Beta program support structure: data collection, customer search and matching, program management, etc. The program was modified to expand the sample size to 50 customers. We had R&D and Test engineers attend as observers. We took the printers to the sites all set up: ink installed, paper loaded, initial power up cycle completed, and had the customers start with the software install. After the install completed, the customer did a quick print and scan functionality check, then we uninstalled and cleaned the customer's system completely of our prototype software. The Iota program was designed to be executed before release of the software with hopes of being able to fix any issues that were found.

The first Iota was an eye-opening experience for everyone involved. It was executed late in the program life cycle, when our existing indicators and measures told us we were ready to ship. Our planned tests were passing at 100% pass rate and our normalized-defect-arrival-rate (nDAR) indicated that we were ready to ship. However, in the live customer environments, our install success rate was only about 50%. The Test lab owned reproduction and characterization of the failures. We only had success in reproducing about 20% of the issues. There were only a few code changes that resulted in directly fixing found issues. In the end, the product shipped with known defects with unknown resolutions.

This was an advantage over earlier programs. Previously, we shipped with unknown defects as evidenced by the continued call center install failure data. With the known defects, work continued even after shipping to help us analyze and characterize the install success and install failure environments. We were able to develop a list of characteristics of customer environments that contribute to the install success or failure. These factors were categorized as:

> PC Hardware factors: computer processor speed, amount of RAM, type of network (if any), and for older system, the size of the hard drive. In recent years, the costs of processors, memory and storage have come down to the point that these are not defining factors like they used to be in the install success space. However, the consumer hardware factors include PCs ranging from circa 1990's hardware to hot-off-the-line netbooks.
>
> PC Environment factors: this includes what type of security applications (anti-virus, firewalls) are installed, OS versioning and maintenance (i.e. system updates maintained), other third party applications and their state of maintenance, and partially installed software that puts the system into an unstable state, to name a few of the things that were verified issues in our customer visits
>
> Customer knowledge and understanding: there are customers who admit that they don't know or understand their wireless network or settings – and they are just fine with a USB connected device. However there are customers that think they know and understand their wireless network and really don't. They will attempt to connect their printer to their neighbor's open wireless access point, or not know what their SSID or passphrase is to their wireless access point. We discovered wireless install success is a major challenge for this group. With the move to enable all our devices to be wirelessly connected, these are major things that we need to address to make sure we can have install success with all our customers.

Test Lab design influences

We had been collecting and analyzing data from the field and customer test programs for several years. We then began looking at ways that the test lab could systematically incorporate the information. The goal was to be able to execute tests in more real-world environments so that we could have multiple iterations of execution. Other goals included facilitating defect reproduction and characterization, and giving development staff access to the environments to assist in debugging and analyzing root cause. These were things that we could not readily do in existing customer test programs. The other driving factor was the prohibitive cost of customer-facing programs. We contract out the customer identification and matching service, we offer monetary compensation to the customers who participate in the programs, and the productivity thrash due to not being able to reproduce and characterize issues in the lab quickly adds up.

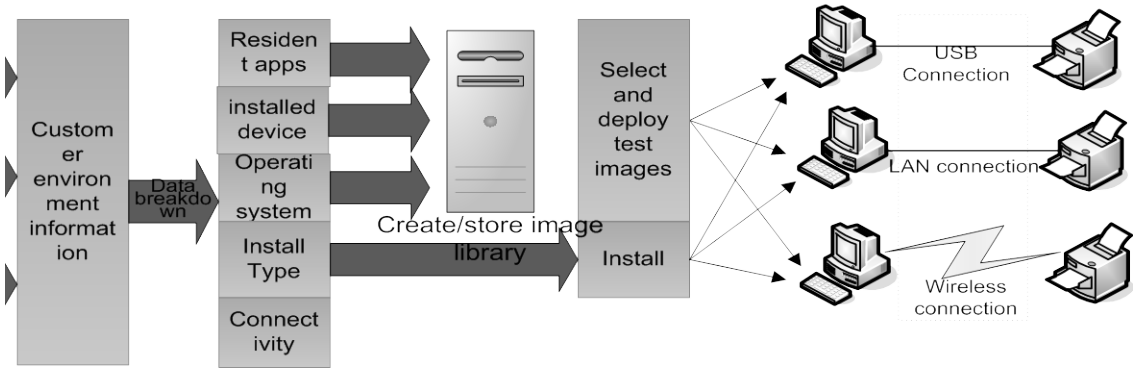
I took the initiative to collate and synthesize all the information collected to propose a new test lab. The over arching lab design is influenced by the desire to have “real customer” environments. This meant securing a variety of PC hardware, creating software environments that were based on real customer combinations versus lab-contrived combinations, securing a variety of wireless access points, and having enough variety in the inventory to be a meaningful contributor to test outcomes. All of this needed to be done with budgetary constraints.

Right Design Objectives

Since I needed budget support, I needed to secure management sponsorship. I made sure that my presentations to management included some core philosophies: the test lab would be customer centric—using “real customer” information to create the lab; targeting replacement of expensive customer visit programs—my goal is to replace the need for the Iota program; and modeling the test execution to include random selection of environments to match the “luck of the draw” aspect of established customer Beta and Iota programs. All of these things resonated with my management and I was given conditional support for a trial run after just one presentation.

To meet these design objectives, I designed a lab with approximately 20 PCs that could have about 10 unique software environments each for a total sample pool of 200 unique PC environments. I also needed an infrastructure to support managing PC images, multiple stations, and multiple wireless access points. The budgetary objectives were facilitated by some good timing and generous sponsorship.

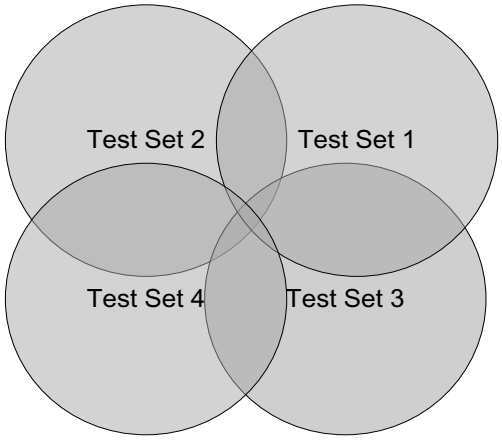
Conceptual diagram of lab design:



Another division of the company was dismantling a hardware lab. I approached the lab manager and secured a variety of PC hardware that was destined to the recycle bin and augmented the collection with standard test lab PCs. The cost for getting my 20 PCs was the cost of shipping them from one lab location to the other. The lab facilities were reconfiguring, which freed up several carts that could hold 4 PCs, mouse, keyboard, monitor, as well as small network and KVM switches. All the peripherals were test lab stock items. So I had the work stations for no cost. Using an existing more powerful test lab PC box, I had a server system built for the cost of added memory and hard disk space. I did some data mining through the data that was collected from customer test program visits (Beta and Iota) and secured system profiles for 200 real customer profiles. We then engaged in the replication of the real customer environments, creating Ghost images stored on the server. Labor cost to do this was the largest expense but, due to timing, was covered by existing budget surplus that needed to be used.

The next phase of the test lab design was to look at how to use the environments such that they could be efficiently used to make meaningful contribution to the overall testing—not a duplication of testing. I looked to the customer test programs for inspiration. I saw that the programs would target new customer pools. We may have a few repeat customer testers, but mostly the customer pools were a random selection of qualified volunteers. I had roughly 200 images to draw on so I chose to define a test run as a set of randomly selected images from the whole pool. This means that no two test cycles are identical. Over the course of the program with multiple test cycles, we would eventually get coverage of all images. The test cycles could be represented as Venn diagrams.

Venn diagram representing test cycles



Once I figured out that I wanted to randomize the environment selection, I needed to define the test set size, the test activities, and what metrics to report. Again, I turned to the customer test programs for inspiration. The Iota customer program was set at 50 installs. This was big enough to get meaningful statistics and about the right size that we could complete a test cycle in a 1-2 week period with one test execution resource. The other thing the Iota program did was to perform a simple print and scan after the installation to verify that everything was as it should be. So my test cycle tasks are relatively simple: install, print, scan. The final thing I wanted to collect and report was the install time—especially since this is a major customer satisfaction goal.

One last test lab design objective was to have a simple report that gave an aggregate picture of the install success in the new lab setting. The report began very simplistically with color coded success percentage across the test cycle set for the separate tasks: install, print, scan from device, scan from software, and average install times. The set was defined at 50 installs, so the averages and percentages reported are across 50 installs. The color coding is based on existing test lab standards: green >90%, yellow 70-90%, red for <70%. This report is referred to as the dashboard.

Sample dashboard (in grayscale):

	Device	Connection Type	Install	Print	Front Panel Scan	Software Scan	Avg.Time	Overall Percentage
Build 1	Device A	Wireless	84% (yellow)	84%	64% (red)	80%	0:27:20	78%
Build 2	Device B	Wireless	100% (green)	100%	86%	96%	0:34:27	96%
Build 3	Device A	Wireless	90%	94%	84%	86%	0:15:38	89%

I have created a web site where all the test data resides. The home page of the site has updated dashboard results for all projects under test. All levels of management have access to the site and email is sent to the program teams when results are posted.

From concept to first test run, the process took about 6 months. The process steps included not only the concept and design work, but also the presentations to management for support, an unplanned office

relocation just as we were finishing design implementation and setup, and securing buy-in for the first program to use the lab.

New Testing Service

The first test was executed for a program that was a newly-architected solution and was the first program to be using agile-type methods for development. They were a guinea pig for new things, so it seemed natural to be part of this new testing service. I was nervous about whether the theoretical lab design principles would actually give results that were meaningful. I waited not-so-patiently as the test execution was done and sifted through the results. It was wonderful when the first failure occurred! It was in an environment that had a “known issue” that our previous installer had addressed but was not included in the requirements for the new installer. It was rewarding to know that the lab design worked as expected.

This led to a core tenet for lab maintenance: if it is a known failure, keep it as part of the lab environment—even for old, unsupported things because our customer install base will have this old technology still out there.

The second test of the testing service came when I was approached by the previous generation software project manager, who wanted to see how the released software performed in the new lab. This was an interesting case because this program had gone through the customer-based Iota test program and there was real customer data available to compare the test lab results against. Again, I was nervous about the outcome. How would the lab results compare to real customer results? In the lab, we could not replicate the customer knowledge aspect, only the technical hardware-software environment. Once again, when the results were reported, the lab and customer tests were statistically identical! Success! The Install Success Test Lab (ISTL) was validated.

There were issues that still needed to be worked out. The program teams pressured me to use the same 50 environments for every test run. Under pressure from management, the program teams set up release criteria that were measured by the ISTL results. The program teams resisted being held accountable for something that seemed like changing environments – they felt that having different environments for each test run was exposing them to an apples-to-oranges comparison. I stood firm on the design principle of randomized image selection for each test run. I pointed out that we do not go to the exact same customers in our customer-based test programs. The outcome of those programs is greatly influenced by “the luck of the draw.” I stressed that I was doing what I could to replicate real customers so the randomization was an essential element to the ISTL results. During program execution, ISTL test runs were scheduled at the end of each sprint – roughly every four weeks. This resulted in multiple test runs of 50 installs for the program. R&D saw the effects of their feature and code changes across the multiple test iterations. The changes generally resulted in improvements in ISTL test results, despite the randomized environments. The development staff became more comfortable with the idea and stopped asking for the same 50 environments.

Another surprising adoption resistance is the pressure to change the environments when issues are discovered to increase install success. This actually comes from my test lab colleagues. For example, I have several systems that are of minimum memory configurations. The test lab generally does little testing in minimum requirement settings due to the productivity hit when using these configurations. When there was an issue with an install failing because of this system configuration, my test lab colleague told me I had to change the system. The system was to specification, the installer was not behaving correctly. It was counted as an install failure in the test run. A defect was filed and the installer code was changed. Another example is when we encounter conflicts with an older application that is not on our official application compatibility list, I was told that I had to remove it from that particular environment because the defect is resolved as will-never-fix so we shouldn’t be testing with this particular application in the environment. I refused because there are customers out there that still have these older applications and we need to know that we are not compatible with them and have support in place to address the issue for our customer.

This has helped solidify another core philosophy for the Install Success Test Lab—sour the pot as much as possible. By this I mean, make sure that I have the known failure environments: 3rd party applications, wireless access points that have caused problems in the past, maximum length passphrases, etc., in the lab at all times. This leads to things like refreshing the stock of wireless access points every three to six months and adding new environments after customer test programs have completed. After a year of lab operation, our environment library is almost 400 images and growing.

As programs worked their way through the ISTL execution, program requirements started being generated from the results. This is one the few test lab programs to generate program requirements. An example of the requirements include install time criteria based on ISTL installs versus the previously standardized "clean OS" install time measurement. This led to a requirement that 80% of installs must complete in x-amount of time as opposed to an average install time. This is because install time is influenced not only by hardware – processor speed, memory, disk space – but also by what other software is installed on the system. In the consumer customer base, there is a vast variety of hardware/software configurations. The previously standardized install measurements did not reflect this reality. But ISTL variety is reflective of customer base, so allowed for a realistic measure. Another example of a requirement driven by ISTL testing is the requirement to be compatible with a wide variety of security application suites.

When ISTL was first conceived, it was targeted for support of the consumer inkjet printers that my test lab supported. Once the results began to be published, there were discussions of the test service in other product labs. To my surprise, I was approached by the laserjet printer R&D and asked if I could support their programs. They wanted to see how their installer products fared in the replicated customer environments. We found several issues that they hadn't encountered in their lab-based testing that increased the quality of these products. The ISTL now executes tests for all the inkjet and laser jet products—even the ones targeted for enterprise markets.

Call Center data examination

Unfortunately, the printers that have been through the ISTL testing service are newly released and have not generated enough call center data to give statistically reliable comparisons at the time of publication. However, the preliminary indications show a significant reduction of install related calls for the ISTL "graduates."

	% Install calls (of total calls)	% Wireless calls (of total calls)
Pre-ISTL	~42%	~30%
ISTL	~28%	*no data

*There is no wireless data because wireless devices have not been released to market at time of publication.

Summary

In conclusion, software test labs can influence product requirements if the test lab offers unique test objectives. In the case of the Install Success Test Lab, there was a very specific and persistent issue that the test lab is addressing. By stepping back and looking at all the attempts to address the issue, then analyzing the faults or failures in the attempts, a different test approach and service was created and accepted by R&D. With a little imagination, creativity, simplistic design and some luck, the testing service is a success.

Another contributing aspect to acceptance and product requirement generation is having well reasoned, data driven philosophies that are adhered to regardless of pressure and influence to change. The consistency in adhering to the philosophies has driven broad organizational changes. This is evidenced by the solicitation across the printer divisions for use of the testing service.

I am waiting for more product introductions to customers and the call center data to be analyzed to see how significantly the test service has affected the install call rates. All the program teams and management feel better about the ISTL “graduates” than we have about any previously released programs. But even then, the backdrop of ever changing environments – operating system upgrades, ever-changing 3rd party applications, web-based computing, etc – are always there to create install success land mines for our device installs. This only means that the Install Success Test Lab will be making continuous adjustments to continue to contribute to the product test cycles.

Customer-Driven Quality

John Ruberto
Intuit, Inc
John_Ruberto@intuit.com
blog.ruberto.com

Abstract

Today, developing software is much more complex than in the past. In this complex world, how do we define quality? We define quality goals with an ever increasing number of "ilities". We perform all kinds of testing: acceptance, performance, functional, concurrency, stress, exploratory, stability, build verification, and finally, regression testing.

Given all of this complexity, it can be difficult as a quality team to define quality. The true definition of quality, however, lies with our ultimate quality consultants, our customers.

One of our values at Intuit is "Customers Define Quality", and this paper shares how we interact with our customers at every stage of the life-cycle to understand their definition of quality. This paper describes the preparation, infrastructure, and practices for the Customer-Driven Quality frame-work.

Biography

John has been developing software in a variety of roles for 25 years. He has held positions ranging from development and test engineer to project, development, and quality manager. He has experience in Aerospace, Telecommunications, and Consumer software industries. Currently, he is the Quality Leader for QuickBooks Online, a web application which helps small business owners manage their money. He received 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.

Prepared for the Pacific Northwest Software Quality Conference, October, 2010

Introduction

A long time ago, in a conference room far away, our QA organization got together for an offsite meeting, a chance to get away from the hustle and bustle of our day-to-day projects. A chance to step back and think about what is important and perhaps think of new ways to improve our quality.

The first exercise as a team was to define quality. We brainstormed in separate teams, writing dozens of yellow sticky notes. Then, as a group we reviewed all of the ideas, grouped similar items into themes, and started pulling together a comprehensive definition.

We ended up with a large pile of definitions; each one seemed valid by itself. One pile was around meeting a number of "Quality Attributes," another pile all about following processes and meeting criteria. Another pile was all about defects (more precisely, the lack of defects). And on and on.

The resulting definition was very comprehensive, and complex. One example, we ended up with 37 different quality attributes[1], and that was just from 1 pile of sticky notes.

The next question, how does this definition help us build better software? Where should we focus our efforts? Some of the definition parts must be more important than others, but which of the 37 quality attributes were part of the vital few? Did we truly have a workable definition, one that would guide us? Well, in asking these questions, we kept coming around to the pile that was all about customers. Some of the ideas were "increased customer satisfaction," "lack of customer support calls," and "high net-promoter." (Net-promoter is a metric we use to measure customers willing to refer their friends to our product)[2].

One slip had written on it, "Customers Define Quality." The reaction was yes, this is obvious, but not too helpful. It has that feeling like "I know it when I see it." It may be true, that our customers will be the ultimate judge of quality, but we only know if they accept it when it's too late, after the project is complete. We needed a definition that helps guide our efforts before, during, and after the project.

However, by thinking "how can we include our customer in each stage of the life-cycle to improve quality," we end up with a number of customer centric practices. We call these practices "Customer-Driven Quality."

Customer-Driven Quality

Customer-Driven Quality is a set of practices for developing software applications to ensure that the product quality meets or exceeds the customer's expectations. Figure 1 shows a graphical view of Customer-Driven Quality.

In Figure 1, customers are the center of the process, surrounded by an iterative software development life-cycle.

Define - Activities related to defining the content of a software project or release. These include ideation, requirements definition, planning, and creating user stories.

Build - Design and Coding phase of software development.

Test - Ensuring the product meets or exceeds the customer's definition of quality.

Support - Helping the customers to use the product and gather feedback to funnel into the next round of product definition.

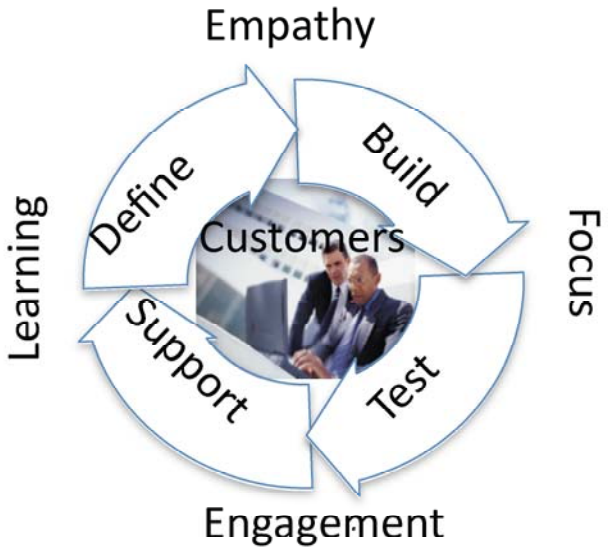


Figure 1. Graphical view of Customer-Driven Quality

Independent of the particular life-cycle stage, Customer-Driven Quality also involves 4 key focus areas. These are shown surrounding the life-cycle, as these activities occur independently from particular projects or life-cycles.

Empathy - Empathy with customers, gaining a deep understanding of the customer's context, including technical skills

Focus - Focus on customers, deciding to invest time and energy in customer-driven activities

Engagement - Direct and unfiltered engagement between customers and the development team

Learning - Understanding actual behavior of customers

Why use Customer-Driven Quality?

Customer-Driven Quality is meant to be applied in addition to, not instead of, all of the traditional software quality practices. Without adding this focus, it can become easy to get wrapped up in the craft of software quality, paying attention to the tools and methodologies, and forgetting why we are developing software in the first place, to satisfy customers.

Looking at the bookshelf, conference programs, and web sites, we tend to pay a lot of attention to the craft of testing and quality. Books, papers, websites, training, certifications, consultants and tools all help with the techniques and methods of building and measuring quality. But, not as much explicit focus is applied to the customer. The following lists shows where customers fall in a word frequency analysis for several software quality publications.

PNSQC Proceedings 2009	Better Software Sept 2009	Software Test and Performance January 2010
test	software	testing
software	testing	software
testing	product	test
team	agile	testers
quality	test	team
code	project	people
time	build	work
data	time	story
product	team	business
development	system	development
system	development	web
process	owner	agile
requirements	work	product
project	quality	use
use	better	performance
work	scrum	career
teams	business	quality
management	code	technology
defects	requirements	code
case	management	company
.	.	.
.	.	.
.	.	.
Customer #159	Customer #510	Customer #32

Table 1, Word Frequency Analysis of several documents, showing the top 20 most frequent words (words like conjunctions, pronouns, names, and articles have been omitted)

This table shows that we frequently discuss the craft of software quality, much more so than discussion customers. Test or testing are the most frequently appearing words, while the word "customer" appears in 159th place in the 2009 edition of the PNSQC Proceedings[3].

Naturally, the word "customer" is #1 in this paper. [4]

Customer-Driven Quality is about getting deeper insights about customer's wants and needs built into our process. It's to build customer empathy in every team member, not just customer representatives.

Looking at software quality literature, there are two rough schools of thought, "old school" and the "new hotness." The "old school" is largely a waterfall process, and is concerned with things like Requirements, Process, Inspection, Verification, Validation, and Traceability.

In "old school" software development, getting customer input is the job for product managers or business analysts. Their job is to understand the customer needs (requirements) and write those in formal documents. Then, the customer comes back into the picture after the code has been released, where the support team is the only interaction. The development team is shielded from the customer by organizational design and practice. This protection is in place to reduce confusion in the development team and protect their time. The developers need to stay focused on implementing the requirements document to maintain end-to-end traceability.

In the “new hotness,” where Agility rules, practices are in place to get more frequent interaction with customers. Practices such as Scrum and User Stories address customer interaction directly, while others like Extreme Programming, Test-Driven Development, Continuous Integration, and Exploratory Testing are focused on the craft of developing software. The iterative nature of these practices guarantees interaction and feedback from customers, or customer representatives.

Customer-Driven Quality is a set of practices that can be used in addition to those already in use. The Customer-Driven practices can be added to any product development life-cycle to supplement your current process.

Preparing For Customer-Driven Quality

This paper describes Customer-Driven Quality practices in two main sections, the “Preparing For...” and a “Life-cycle view.” The “Preparing For...” section describes a set of practices that can be used independent of a project life-cycle. These practices provide a foundation for Customer-Driven Quality.

Team Composition and Focus

Everything starts with the team building your software. Without the team, the software wouldn’t exist to delight customers. So, efforts to prepare for Customer-Driven Quality should start with the team.

In addition to technical, process, and leadership skills, your team should also have some customer advocacy mindset built in. A very fertile area to recruit customer advocates is from the customer care organization. What they **may** lack in automation or coding skills they compensate with deep knowledge and empathy for your customers. In the customer care role, they were talking all day directly with customers, helping them through problems or training them in how to use your software. Who better to help build the next generation of your products than the people who supported the previous generation?

At Intuit, we have a win-win scenario for including customer care people in the development life-cycle. Many of our products are seasonable in nature. For example, the vast majority of our customers use our tax products during a few months of the year. This seasonable demand for care agents provides an opportunity during the off season, the time when we are building next year’s product. We are able to supplement the test organization with the best support agents testing the product. This levels the staffing needs of the whole company, while improving quality of the next year’s product.

At release time, the support agents return to their support roles already well trained in the product. This is also provides a career path for individuals in the customer-care organization. Many of our best quality analysts and engineers took this path to the QA organization.

Goal Setting

Organizations differ in the degree of formality of Performance Management systems. Some have very formal goals that are frequently updated, others are more informal. Regardless of the culture of goal setting or monitoring progress towards these goals, the activities associated with Customer-Driven Quality should play a large role.

Everyone in the organization can participate in the customer-driven activities. Many have customer interactions as an explicit part of their job, while others need to take the initiative to engage in customer interaction. I expect everyone on the team to participate, even the test automation framework developers.

One of the most powerful ways to influence any activity is to model that behavior yourself. If your team sees you investing your time talking to customers and gaining insights from that activity, they are very likely to participate as well.

Building Empathy

Most software products are developed for “real people” not software developers. Sometimes those of use that are creating software every day, using technology in every aspect of our lives and are savvy computer users sometimes forget that our customers don’t have the same skills and thought processes.

For example, I’m working on the QuickBooks Online team, a Software-As-A-Service (SAAS) offering to help small business owners manage the money aspects of their business. Our customers are focused on their craft, whether it is running a restaurant, an independent bookstore, a medical practice, or an automobile service station. They are not working with a variety of browsers and operating systems. They may not think to open the preferences to see what options are available. They often select the default choice presented in our UI. In short, they are experts at their field, and expect us to be experts at ours.

These are a few practices that have proven useful for building empathy with our customers. For learning how they think, and what they think about, and what they expect to be taken care of for them.

Customer Care rotation for new engineers

All new college hires go through a rotational program designed to start their career with customer empathy, by spending a couple of months in Customer Care. In this program, they are provided the same training that our care agents receive then get on the phone to help our customers. The learnings are amazing. After 4 years of academic learning, steeped in algorithms and theory, they get exposed to the day-to-day issues that our customers encounter. An example of an insight that an engineer passed on to me, “I was amazed how many people didn’t know about the right-click menu.”

The Customer Care rotation program is outstanding, but it’s also pretty expensive. It’s mostly been applied to new college graduates, and it’s combined with a more general orientation to our company and corporate life. Another method that is used by experienced professionals as well as long time employees is to listen into customer calls.

Listening to calls

Many of our customers call in for support. As we all know, “those calls may be monitored for quality control purposes.” The monitoring infrastructure provides an outstanding opportunity to get engineers and testers exposed to these calls. In fact, listening to both sides of the conversation while not an active participant gives some deeper insight to the customer’s issue and how we resolve those issues.

Many of our staff meetings start with listening to a call, followed by a discussion about the insights each of us learned by the process.

Follow Me Home

The Follow Me Home process started a long time ago, when the main distribution channel for our products was a retail store. Engineers and marketers would hang out at a local retailer and watch people shop for our products. Marketers were interested in how the brand new customer chose our product from the alternatives on the shelf, but the fun stuff happened when they “followed” the customer home.

The team would approach the customer and ask to follow them to their home or place of business and just observe the first time use process. Did they read the instructions? Have trouble installing? Accept defaults or investigate each option?

The process continues, but is usually set up ahead of time via email contact. It’s very instructive to observe the customer use our products, in their environment. You notice things that are not spoken or written in surveys or support tickets.

One example insight, on a follow me home that I conducted with a customer that had complained about our service being slow. We checked the logs; all the transactions were in normal tolerances. We checked her computer, network connection bandwidth, network latency to our data center, etc. No slowness there.

In observing her, though, she had a stack of bills that needed to be paid. This stack was on a credenza behind her desk. She would take a bill, swivel her chair around and enter the data on our page. Then hit save, swivel and get another bill. She said "there, see, the screen is not yet ready to enter a new bill. It used to be ready by time I swung back." Ah, we now have a new definition on how fast is fast enough: it can't impede her work flow. (The real insight, why are we making this poor woman type in information, the bills should be automatically imported.)

Usability Studies

Web design is about a lot more than fonts and style sheets. Ultimately, it's about ease of use. To test ease of use, and gain insights about customer behavior, the UI designers invest a lot of time in formal usability studies. They will invite customer (or potential customers) in house and observe them using our product or prototypes. This is an excellent opportunity to get an engineer or tester some insights as well. Make friends with the design team, and get some invitations to observe as well.

Voice of the Customer

Our customer care team handles many calls, emails, and online chat sessions each week. They produce a call drivers report, which lists the top 10 issues that generate a support call, with the goal of fixing product or process so the call is not required. This aggregated and abstracted data helps focus priorities for product changes, and the raw data that feeds these reports are a great resource for gaining insights about our customers. We call this raw data the Voice of the Customer.

Our support system has a feature that pulls out 50 random issue summaries, and emails it to the entire team. The first item on our weekly development staff meeting is to review the call drivers and discuss insights learned from reading the verbatim comments.

Net Promoter Call Backs

Our marketing team conducts a customer satisfaction survey every quarter, called the Net Promoter survey. (Net Promoter is a measure of the willingness of our customers to recommend our product to their friends and colleagues). Part of this process is to call every customer that is a detractor (not willing to recommend our product), and a random sample of the promoters.

The marketing team loves getting help in making these follow-up calls. The team that builds and tests the product is an excellent choice for this task. Each engineer and tester is expected to make 3 of these calls per year. It's not a huge investment in time and the insights gained are very valuable.

Product Challenge

One way to understand our customers is to act like one. We hold frequent "challenges" where we put ourselves in the shoes of our customers and try to use our product. This is a great way to introduce new team members to the product.

In QuickBooks, we have a virtual "shoebox" of receipts and invoices that a customer may have when they decide to get better organized and buy our product. The challenge participants are given this shoebox, plus a CDROM of our software, and asked to produce financial reports and some insights about where the money is going.

Infrastructure

Preparing for Customer-Driven Quality also involves some tools and organizational infrastructure that most likely already exists in software development organizations. Here are a few of the main components.

Customer Care Organization

The customer care organization has already been mentioned many times in this paper. It's extremely important to build relationships with the care organization and have a deep engagement with them.

Feedback Mechanisms

Besides the customer care channel, it's very useful to get direct feedback from customers in the context of using our products. We've build a feedback widget that can be placed on any of our pages.

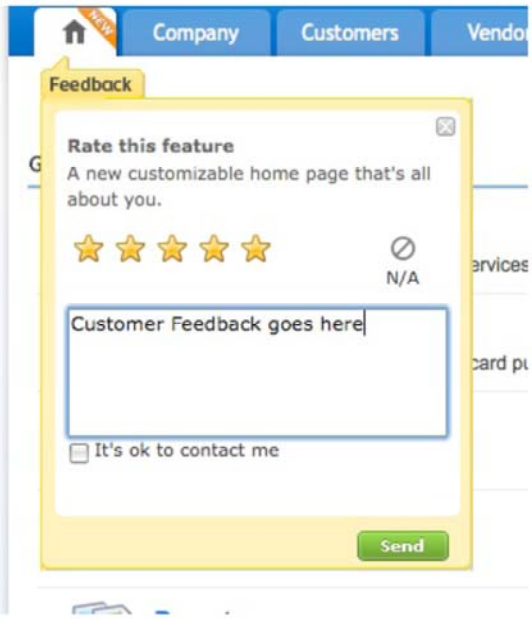


Figure 2. Screen shot of a feedback widget, which allows customers to provide feedback directly for the feature that is currently active on their window.

The feedback widget allows our customers to give the feature a rating, from 1 to 5 stars, and to type in some comments, and optionally contact information if they are willing to have a conversation.

The engineers and testers who created the feature monitor the feedback. This has been an extremely useful mechanism to learn what is working, and find opportunities to improve the feature.

Here is some example feedback, where customers provide valuable insights in how we can make the feature even more useful. [5]

Rating	Customer Comments
4	I love this feature! But I would like to be able to view notes like the old view allowed.
4	I would like to be able to do a statement for an individual client from this screen. Like we can do now.
4	Just started using this screen. Looks great so far.

Figure 3. Screen shot of a typical sample of customer feedback.

Another example, in the View My Paycheck service, the quality team was so passionate to get customer feedback, that the quality engineer did the coding necessary to integrate a third party feedback service.

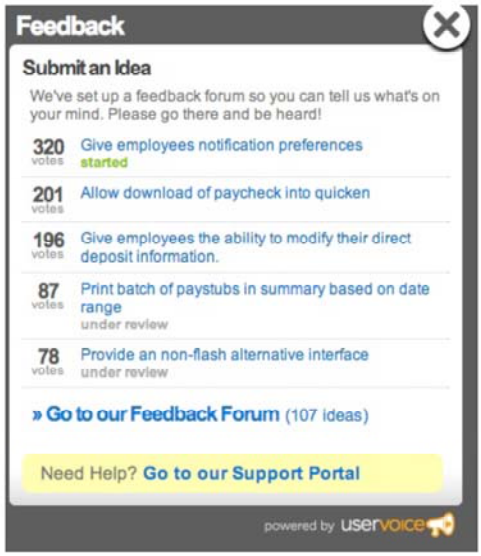


Figure 4. Screen shot shows another example of a customer feedback mechanism used in product. This one allows customers to vote on other customer's feedback, helping to identify which of the new requests are most popular with existing customers. [6]

Text Analytics

Much of the feedback is in text form. The feedback widgets, support emails, support chat transcripts, and call summaries. While useful to read the raw input to build empathy, it's also important to aggregate the raw data to gain insights on the most important issues faced by your customers.

We are lucky to have a lot of customers, so we get a lot of feedback. Automating the analysis of feedback is efficient and gives an opportunity to analyze it more frequently.

A tool that we've found to be very useful is Sentiment Analysis. Sentiment analysis is a semi-automated process that measures both the frequency of occurrence of each issue type, but also measures the intensity.

Text frequency analysis allows us to count the number of times our customers have a particular type of problem (i.e. how many calls are about login). Where sentiment analysis shines is adding customers intensity. For example, "I really hate your login" is generally worse than "Your login is difficult."

Monitoring

Our software has many monitoring mechanisms built in to make sure its operating properly and to alert the support team in the event things start to go wrong. There are action logs, exception logs, and performance monitors. Each of these logs also can provide some insights into actual customer behavior and experience.

The QA teams learn what logs are available, how to get access, and how to analyze these logs. One example where we used the log files, we pulled the action logs across 30 days of use, and identified the top 10 transactions. This information helped us to prioritize the test automation program.

Engage with Customers

The incoming support calls, feedback widgets, and customer usage logs are great ways to get information about your customers, but there are a lot more opportunities to engage directly with customers.

Social Media

Product specific blogs allow conversations with customers. New features can be announced, articles written to help customers, and conversations happen through the comment feature of popular blog engines. One example where we used blogs effectively in QuickBooks Online (http://blog.quickbooksonline.com) was to influence out customers to upgrade their browser. One quality constraint that we were living with was the support for Internet Explorer 6. A lot of special code had to be written for IE6, and testing multiple browsers reduced the amount of test on any specific browser. Newer alternatives had better security, faster performance, and better compatibility with standards.

We wrote a series of articles on the blog to influence our customers to upgrade. Each time a new article was published, we saw a reduction in the usage in the web analytics logs. Influencing our customers to take some action at the client side is an example of customer-driven quality.

Facebook and Twitter are other venues for learning about customer's usage of our products, and an opportunity to communicate with them. Often, customers are posting on these sites their frustrations and joys with our products. It's pretty simple to have an agent providing real time search for our product name.

Search Engine Strategies

We also use search engines (Google, Bing) to search for our products and what our customers are potentially saying online about our products. Doing this exposed several community forums dedicated to using our product.

Customers will post their likes and dislikes about our products, questions about how to perform a task, and desired new features. This is a good way to gauge the customer's opinion of our quality.

The search engines have a feature called alerts, which will automatically perform the search specified, and email the results in either a daily digest, or real time. To monitor the "blogosphere," we've set up alerts with the popular search engines.

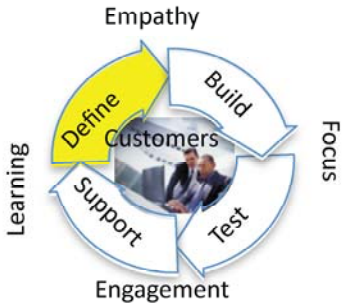
Customer-Driven Quality: Life-cycle View

The previous sections described a number of practices that provide the foundation for Customer-Driven Quality. This section provides a simplified product development life-cycle and describes the associated customer-driven practices.

This simple life-cycle is described in 4 stages: Define, Build, Test, and Support.

Define

The Define life-cycle stage is where the product requirements and scope are defined. The customer-driven practices are intended to help build the right product for our customers.



Investment level

One of the most powerful decisions that can be made during the definition phase is deciding how much of the limited development bandwidth to apply towards three broad categories: product infrastructure, solving current customer pain points, and building new value for current or future customers.

These questions are not easy to make, but each category must be considered. Often, the product definition phase is dominated by building value for new customers, but product infrastructure and current customer pain points are important as well.

The product infrastructure must be maintained and improved to prevent future pain points, such as performance or availability issues. Solving current customer pain points are one of the reasons we gather and analyze all of the feedback.

A few ways of managing this prioritizing the current customer activities include:

- Applying a "tax" of development time before considering new features; keeping some development resources in reserve so they can concentrate on satisfying current customers.
- Managing requirements and features in a common prioritized backlog along with infrastructure and customer issue. Having a common repository will help the team make these tradeoffs.
- Periodically, a "customer love" or "net promoter" release can be useful, where the entire focus is just solving pain points.

Ideation Process

Customer-Driven Quality does have a natural predator, the HIPPO. Not the aquatic mammal from Africa, but the "Highly Paid Person's Opinion". The HIPPO is a term we use to remind ourselves that it's the customer's opinion that counts, not the boss's.

Instead of building the feature set that is desired by the influential people in the organization, instead the customers should decide what we build. We call this process the Ideation Process.

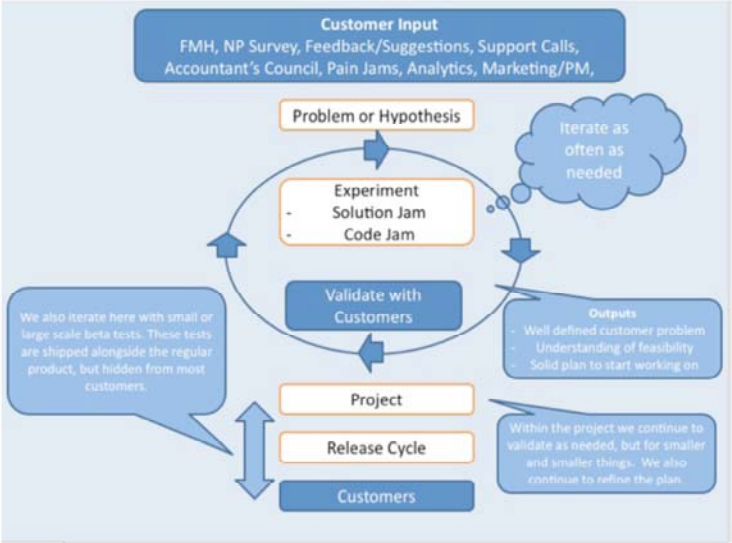


Figure 5. The "Ideation" process, where features are defined while working closely with a set of customers.

The Ideation Process involves understanding the customer's pain points, brainstorming ideas to solve the pain, and testing the ideas in the market with real customers. To emphasize the experimental nature of this process, the ideas are called hypotheses.

In the center of the diagram of Figure 5, the core of the Ideation process is a Solution Jam & Validation with customers. The product team brainstorms solutions with customers, then will work with designers or engineers to produce a lightweight prototype (paper or a very quick Flash application) to validate the solution. Solutions that are deemed correct then go to the next stage of validation.

Since the prototyping is done with a limited set of customers, and these customers know they are part of the experiment, the solution then goes to the next stage, testing in market.

The prototype and list of user stories that result are the input to a product development cycle to produce a working product. This working feature is then put into the next round of testing, where its exposed to actual customers. These experimental features are exposed to several dozen to several thousand customers. Feedback and analytics are monitored closely to judge the effectiveness of the solution. Winning solutions become part of the product and are widely deployed. Others are either quietly abandoned before getting too much exposure or the team goes back to the drawing board.

The Ideation process can be thought of as a product management process, and being tangentially related to "quality." However, a better way to think about it is testing the requirements before the product is built.

User Stories not Requirements

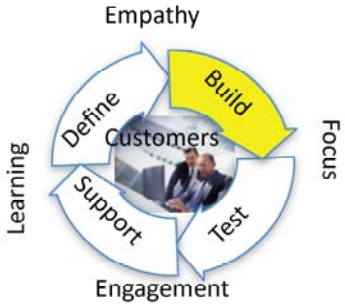
In the traditional model of software development, a special team of people, called business analysts or product managers, talk to customers, understand their needs, and document those needs in a formalized language. This formal language is called requirements.

Great care is taken to write the requirements in a manner that facilitates traceability, completeness, and abstracts any hint of implementation. When reading these formal requirements, it's frequently easy to miss the point completely.

Documenting the product requirements in the form of User Stories helps keep the focus on customers, and removes an opportunity for ambiguity. User Stories can also be read and understood by customers.

Build

The build stage of the life-cycle is where design and coding takes place.



Personalized Development

We have several methods for developers to interact directly with customers during the implementation phase. These boil down to putting a name, face, and engagement with actual customers to build the right product in the right way for our customers.

The Adopt a Customer approach has a developer choosing a small group of customers. The developers then build the feature collaborating directly with the customers. The developers have access to customers to ask questions, give frequent informal demonstrations, and brainstorm ideas. This process provides focused problem solving, rather than guessing intent from the requirements (or asking HIPPOs).

We maintain a list of customers willing to provide frequent interactions, through the Inner Circle program. Customers opt in to this program, and are available for consultations.

Default Behavior

One customer behavior that has been learned through web analytics is that the vast majority of customers do not change the default choice when provided multiple choices. This behavior drives many of the design decisions. For example, a user interface design where the customer should choose from multiple options (State, for example), should not pre-select a default, especially if the list if alphabetical.

Reviewing the default options is a specific line item in the design review checklist.

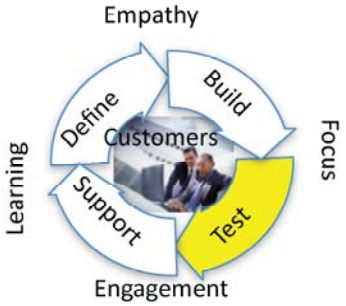
Build with Customer's Platform

Many self inflicted errors come from cross platform development. To the degree possible, this should be avoided. The product should be built on the platform used by customers.

In the current state of web application development, most developers prefer to develop with Firefox, because of the superior development and debugging plug-ins. These plug-ins were developed by developers, for developers (talk about Customer-Driven Quality). However, most customers use Internet Explorer as their browsers. Ideally, the web UI should be developed with Internet Explorer, or at least tested by the developers before they check in.

Test

The test phase is where we ensure that the product meets the defined requirements and is suitable for release.



Beta Test

Releasing product as a beta is a time tested practice to include customers' feedback in the development process. One observation about beta test, some customers are not willing to test a beta product, so we may miss out on their feedback. One benefit of an iterative development cycle, we always have a chance to include customer's feedback.

Rolling Deployment

In the first day after fully deploying our application, we see several orders of magnitude more product usage than in the entirety of our testing cycle. To make sure we have a smooth deployment, we deploy in stages. This helps us ensure we didn't miss anything in our development process.

We deploy to approximately 10% of our customer base the first evening. Then, we monitor the feedback closely from these customers, looking for any new or unique issues being reported by these customers. The balance of customers see the new release a couple of days later. This process gives us the chance to correct any bugs that were missed by the test team.

Analytics Driven Test Plans

Several members of the QA team are also participants on the Web Analytics team. The Web Analytics team is largely used by the marketing team to understand customer's behavior for the purposes of finding opportunities to optimize customer value, but we've found a lot of useful information to inform the test strategy.

One example where this added value, we have an automated test suite that duplicates the top 12 customer workflows. This test suite is executed every day.

Performance Testing in the Real World

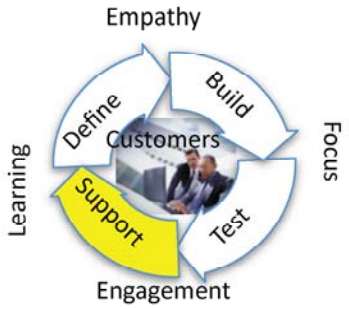
Performance and scalability testing is both supplemented and calibrated by using a remote testing service. The remote testing service executes test scripts from many locations across the world, which provides information on how our application is running from our customer's perspective, not just from inside our firewall.

The service that we use allows us to test performance from many locations in the world, both at network backbone locations and on actual customer's machines, which may be connected by dial-up, DSL, or Cable modem.

This service has helped us optimize performance using Content Delivery Networks, and optimizing content download time.

Support

Since this is a circular life-cycle, there is always customer learning that can be applied to future iterations.



The Customer Care organization is all about interacting and satisfying customers. This section will provide a few ideas to incorporate the support phase in defining quality for future iterations.

Development Team Support

When we develop and release new features of significant complexity, a very useful process is to include the development team on the front line for customer care. This practice helps by supplementing the customer care team, as the call load is likely to be greater for a new feature that is not yet understood. By putting the developers in the support role, they get that direct interaction which helps them tweak the design.

Analytics & Feedback

In addition to the feedback channels mentioned above, the support team is an excellent source of knowledge. Since they interact with customers as their main task, they are able to give customer focused insights to the development team.

Social Media

Interacting with customers through social media, like Twitter and Facebook, were mentioned earlier as way's to build empathy with customers and discover their definition of quality. Likewise, it's a great way to interact with customers in a support capacity. Customers will frequently post their problems to their friends. Replying directly, proactively, and helping solve the problem usually results in a positive experience for the customer.

Conclusions

Customer-Driven Quality has helped our team provide focus on what is truly important, and help cut through the clutter of complexity facing our software development decisions. The following checklist distills many of the practices of Customer-Driven Quality:

Customer-Driven Quality Checklist
Does your team have a customer advocate?
Do your team members communicate directly with customers?
Are Customer-Driven Quality activities part of the goals for your team?
Is the support case feedback available to the development team?
Does your team regularly analyze and act upon the support case feedback?
Are members of the team engaged with customers on social media? (blogs, forums, Twitter, Facebook)
Are you searching for new mentions of your product on a regular basis?
How much effort is being allocated to delighting current customers?
Are requirements determined by actual interactions with customers or by industry trends?
Do you track actual usage, and use that information in your quality strategy?
How does the development platform differ from a typical customer's platform?
Is customer acceptance testing performed with actual customers?
Are you calibrating your test/quality strategy based on actual customer metrics (usage analytics, performance, platforms, etc.)

Notes and References

[1] Chithambaram, Raju, Non-Functional Quality Attributes, Intuit Life-time Design Template, April 2010

[2] Net Promoter is a customer satisfaction metric, where customers are asked their likelihood to recommend a product or service to their friends. The responses are classified as "Promoters" or "Detractors." The metric is calculated by subtracting the percentage of detractors from the percentage of promotors. For more information on Net Promotor see http://en.wikipedia.org/wiki/Net_Promoter or http://www.netpromoter.com

[3] The text frequency analysis used 3 documents, the Conference Proceedings of the 2009 Pacific Northwest Software Quality Conference, the September 2009 edition of Better Software Magazine, and the January 2010 edition of Software Performance and Test Magazine. The text frequency analysis was performed by converting the PDF documents to text, and using a word counter program. Words like articles, pronouns, conjunctions, and proper names were eliminated, as these were not relevant.

[4] This document was analyzed using the same process as the other documents, and the top 10 words in descending order are: customer, quality, product, team, development, software, driven, process, test, and support.

[5] This example screenshot is taken from QuickBase, a Software As A Service (SAAS) database offering which allows easy database creation.

[6] This example uses another service, UserVoice.

Incorporating user scenarios in test Design

April Ritscher
Senior Test Engineer
Microsoft Corp.
aprilri@microsoft.com

Abstract

How many of us have tested an application and certified that it met the requirements stated in the functional specification, only to find out that it does not meet the business need? Many times as software test engineers we are brought on to the project after the requirements have been gathered. This gives us very little visibility into the early discussions on what the user needs to accomplish with an application solution.

As part of our work to improve the software testing process for our group (LCAIT), we have been looking into ways to move upstream in the process and ensure that we add business value by testing our applications based on user scenarios that reflect the usage of the application. This required us to change our approach in writing test cases and changing our focus from the traditional functional testing to user scenario focused testing. To achieve that objective we used visual representations of possible user actions and application responses that align with the user scenarios.

This paper will describe how we break out our functionality into individual test cases and use these as building blocks to test the end to end user scenarios. It will also describe how we extract information from the flowcharts to be used during manual and automated testing.

Biography

April Ritscher has been a senior test engineer at Microsoft for the last 10 years. As test lead, she has worked on a variety of different projects including a project for Bill Gates. She has received several awards including 5 Ship-it awards and 3 IT Pro awards. She has worked on several global teams with both India and China. She is very passionate about quality and always looking for ways to improve the software testing process.

1. Introduction

As test engineers we are asked to ensure the quality objectives of an application are met. In the past we have accomplished this by comparing the functionality developed against the functional spec and certifying that it matched. Now if you asked someone in IT how we feel we have delivered against customer expectations you would hear that we feel we meet their expectations 94% of the time. Now if you ask our customers, they would say we meet their expectations 48% of the time. How can this be? If we certified that our application met the functional spec how can we have such a large discrepancy between our assessment and that of our customers?

The problem we have found is that the customers assess the application based on whether it allows them to accomplish their day to day activities efficiently and easily regardless of what was defined in the functional specification.

What we found is that using the Functional specification or Requirements as your measuring device does not always ensure that the application will meet the user's needs. Instead we need to move upstream in the process and find out what the customer needs to be able to accomplish in an application agnostic way. The best way to do this is to gather user scenarios at the beginning of a project and incorporate these all the way through the project life cycle.

2. Goals

In order to shift from the mind set of being a test team that merely certifies an application to a Quality assurance team that influences the design means that we need to move upstream in the process. One of the main problems that we have in test is that we get involved at the end of a project. According to Authors Roger S. Pressman and Robert B. Grady the cost to fix software defects varies according to how far along you are in the cycle. As you can see from the figure below, issues found later in the project are significantly more expensive than those found in earlier phases.

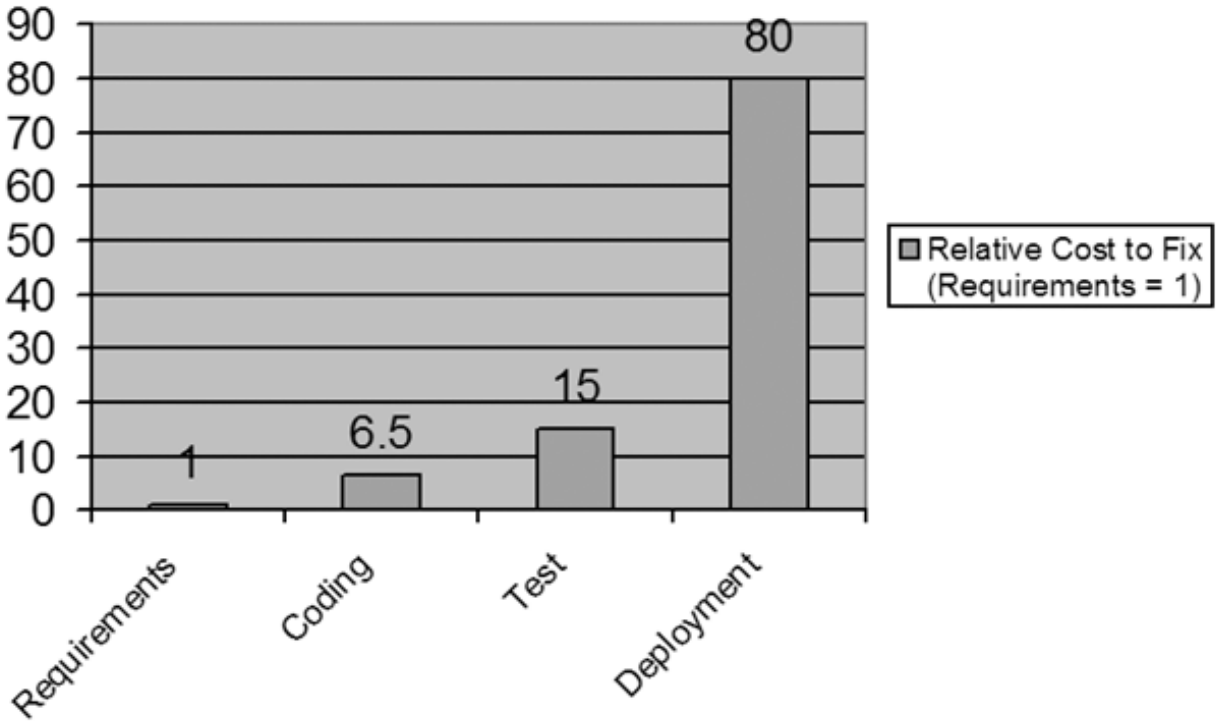


Figure 1: Variations in costs to fix a software defect

By involving the test team earlier in the project we are able to increase our capacity as well as become more influential on application design.

Another way to increase the capacity of the test team is through Automation. One of the major deterrents to automation is the cost. One of the goals we had during this approach was to find ways to reduce the amount of maintenance cost needed on our automation code.

Finally our overall goal of our process was to ensure that any solution we delivered met the business needs and that we didn't end up in a situation where we coded per the specification but didn't meet the user expectations.

3. What is a User Scenario?

The first question you may ask is what is a user scenario? A scenario is a concrete narrative story told from the customer's point of view that explains their situation and what they want to achieve. A well written scenario will be application agnostic and will not say how they will accomplish their goals but will focus on what those goals are. It should also describe the customer "dream state" of what they would like to be able to do, that they may not be able to do today with existing tools or applications. A scenario is real world descriptions which often span more than one feature.

Example of a scenario:

> Jane is at the airport at 9am waiting for her flight to Chicago for a business trip, leaving at 10:30am. While waiting for her flight, Jane notices an important message from her colleague. The presentation in Chicago has been moved to a new location, and the partner has asked to see 3rd quarter projections as well as 2nd quarter. Jane responds to her colleague to let her know that she got the message but doesn't have the 3rd quarter projections on her computer – can Sue send it to her immediately, before her flight leaves? A couple of minutes later, Jane gets the 3rd quarter projections and has just enough time to review them before her flight. She is relieved and confident that she will be fully prepared for her meeting.

4. Benefits of testing with user scenarios

How do Scenarios help us? A well written scenario allows you to see your solution through the user's eyes and should paint a vivid picture of their user experience. It also helps to identify which features will be critical to the user's day to day business process and reduce the risk that these could be inadvertently dropped out of scope. They also help to identify design gaps earlier in the design phase. By comparing your application design to your user scenarios you can easily see whether the proposed solution will help or hamper their ability to accomplish their goals.

Using scenarios during the test phase also helps you to prioritize your defects. If a bug is found that prevents the user from accomplishing their goals laid out in the scenario this becomes higher priority than other bugs that may not be part of the user's main workflow.

5. From Conventional To Innovative

In order to incorporate user scenarios in to our testing we needed to shift our mind set on the definition and representation of a test case. Typically in the past we had created functional test cases which were text representations that included all the steps you needed to execute in order to test a certain feature. The problem with this is that we ended up looking at functionality in isolation as opposed to looking at how the user would use the application to accomplish their goals. Also we needed to have thousands of these functional test cases to completely test all of the application features. Reviewing test cases can be a mind numbing exercise and if you have thousands of them, the chances of getting meaningful feedback from your reviews is slim to none.

Instead we decided to visually represent our test cases in Visio in order to facilitate reviews and ensure that our test cases aligned with the user scenarios. Why Visio? We chose to use Visio because it was readily available to the entire team since everyone already had office installed. We also needed a tool that would allow us to export our workflows in a textual form to be integrated with our test case management tool. However which tool you use is not as important as the fact that you visually represent your workflows.

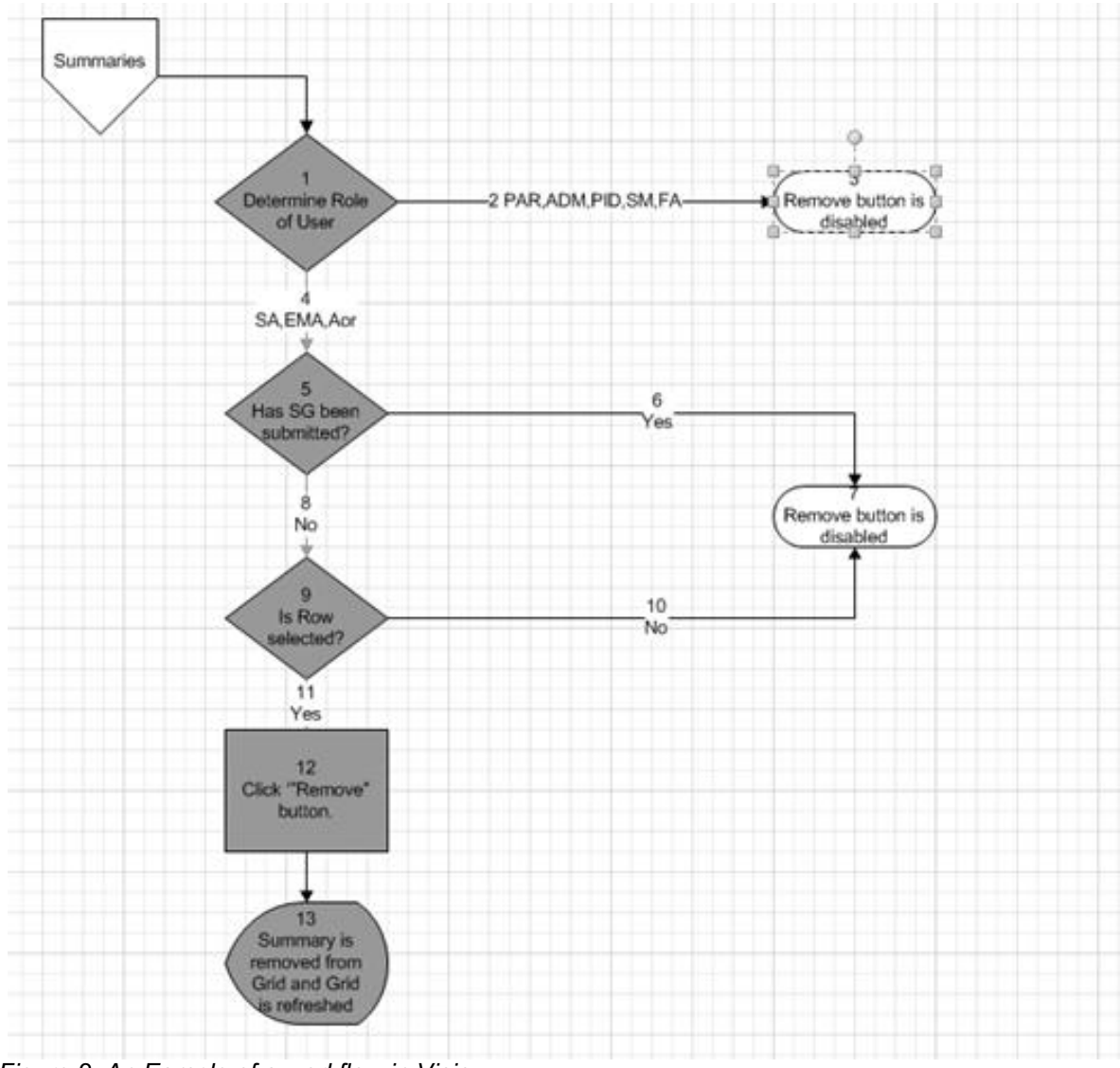


Figure 2: An Eample of a workflow in Visio

Another area of focus we had was on automation. One of the challenges we had in the past was that our automation was framework dependent and any decision to move to a different automation framework would require an additional automation cost. Instead we transitioned to a model using C# that allows us to use any framework we chose as long as the appropriate classes are public.

Finally our last area of focus was on the way we incorporated data into our automation testing. Typically in the past, if we had multiple data variations for a given test case, we duplicated this test cases multiple times. The problem with this is that there was a lot of redundant data which resulted in a much higher maintenance cost. Instead we extracted the data variations in to their own entity that allows us to execute a single test case against multiple variations. In our line of business most of the business logic is very data dependent. By extracting this information we were able to reduce the number of test cases by 88%.

You can see from the figure below how the three areas of focus map to our original goals.

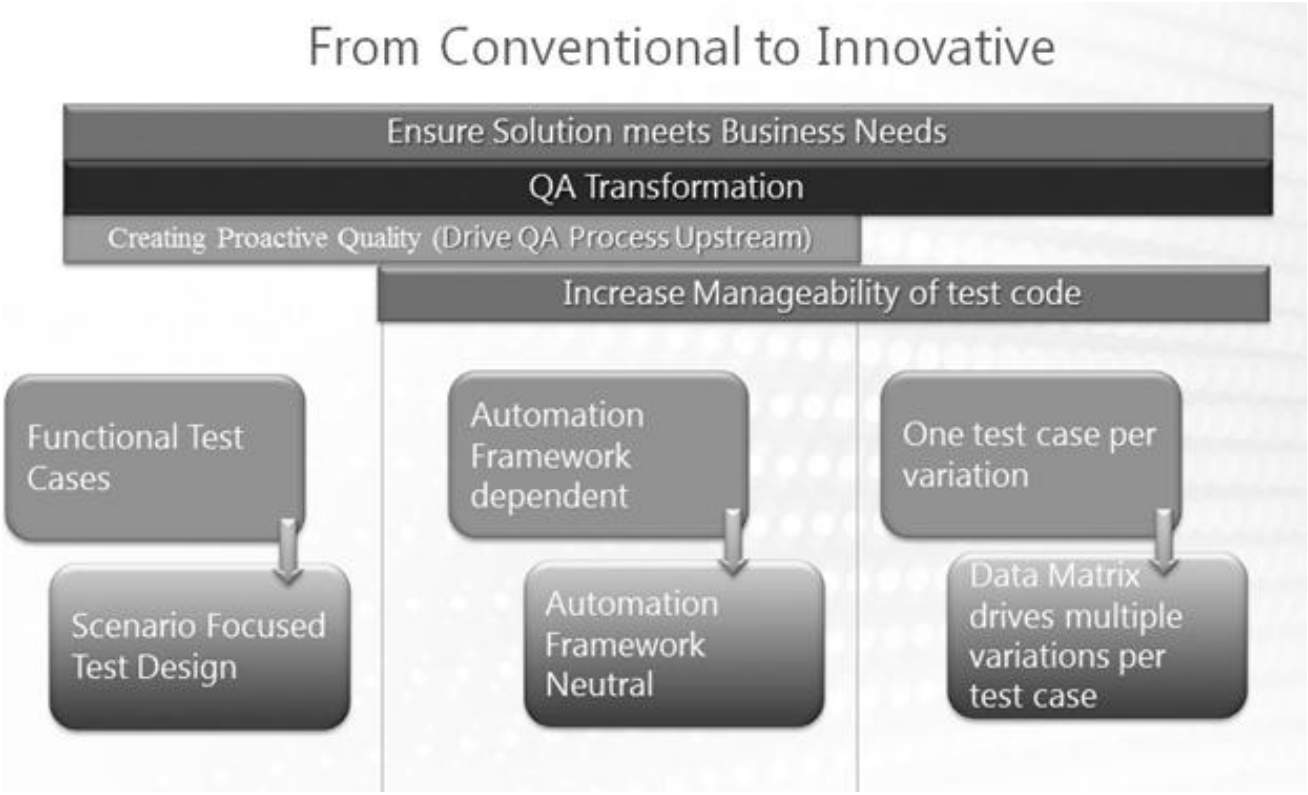


Figure 3: Aligning our focus areas to our Goals

6. Scenario Focused Test Design:

How do we pull it all together? In the figure below you can see how we incorporated user scenarios in to our testing process.

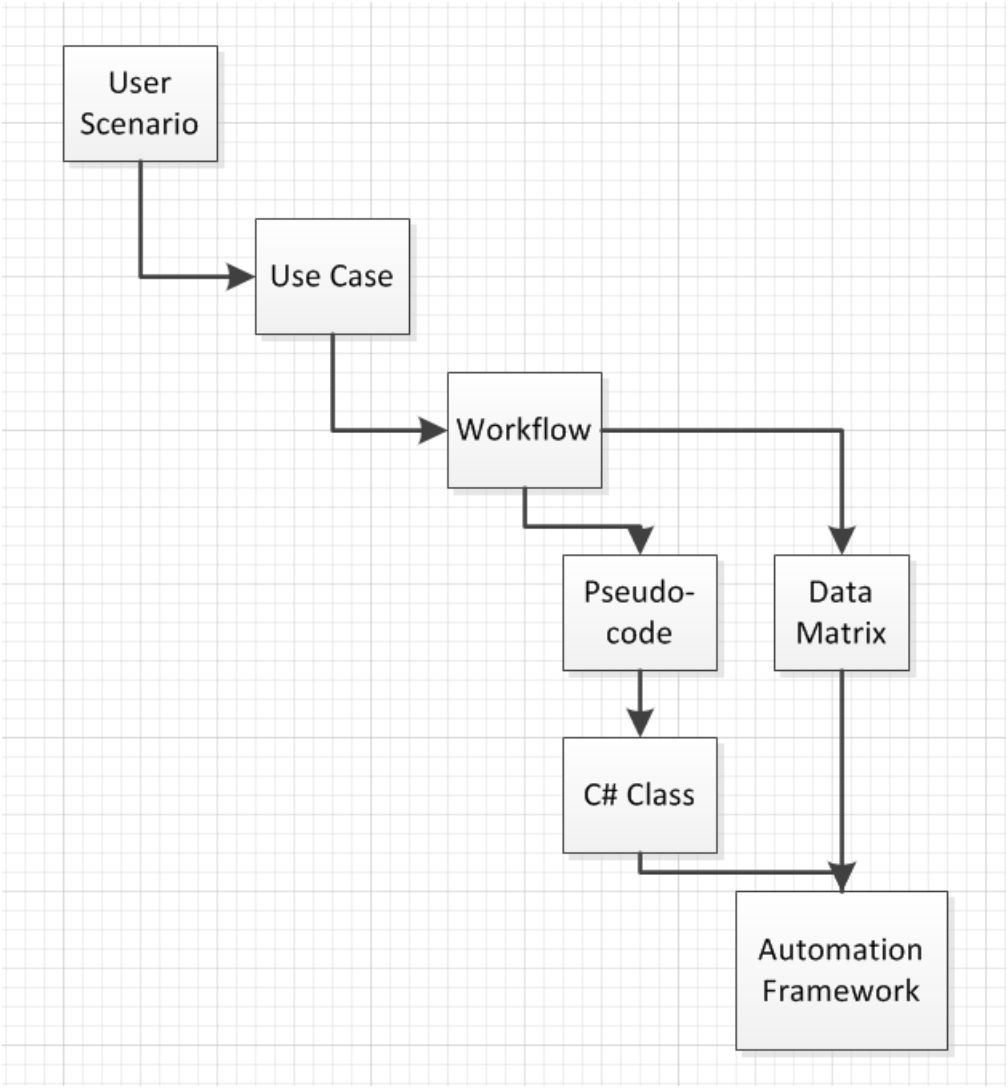


Figure 4: User scenario focused testing workflow

User Scenario: We start with the user scenario which describes what the user wants to be able to accomplish.

Use Case: The User scenario is then translated in to a high level flow in Visio that corresponds to a use case

Workflow: We then take the use case and expand on it including information from the functional spec on how the application is expected to respond the user actions. We document their main flow as well as any alternate flows that they may need to accomplish.

We use tabs in Visio to compartmentalize our functionality into manageable functions that can be strung together to align with a variety of different user scenarios. We simplified the flowchart process by using only 6 basic shapes in Visio. The first type of shape is a user step that defines what the user will do. The second shape is an application response. The third shape is a decision that represents any time there is a possible branch in the workflow. The fourth shape is a dynamic connector which is used to tie the shapes together. The fifth shape is an off-page connector which is used to tie the different tabs together. The final shape is a terminator which indicates when the user has reached the end of a particular function and should be returned to the originating function.

Using built in functionality in Visio we associate different meta-data to these shapes. This includes whether this step represents the main or alternate flow, any conditional hints like the beginning of an IF\Else block and the function name that corresponds to the tab name.

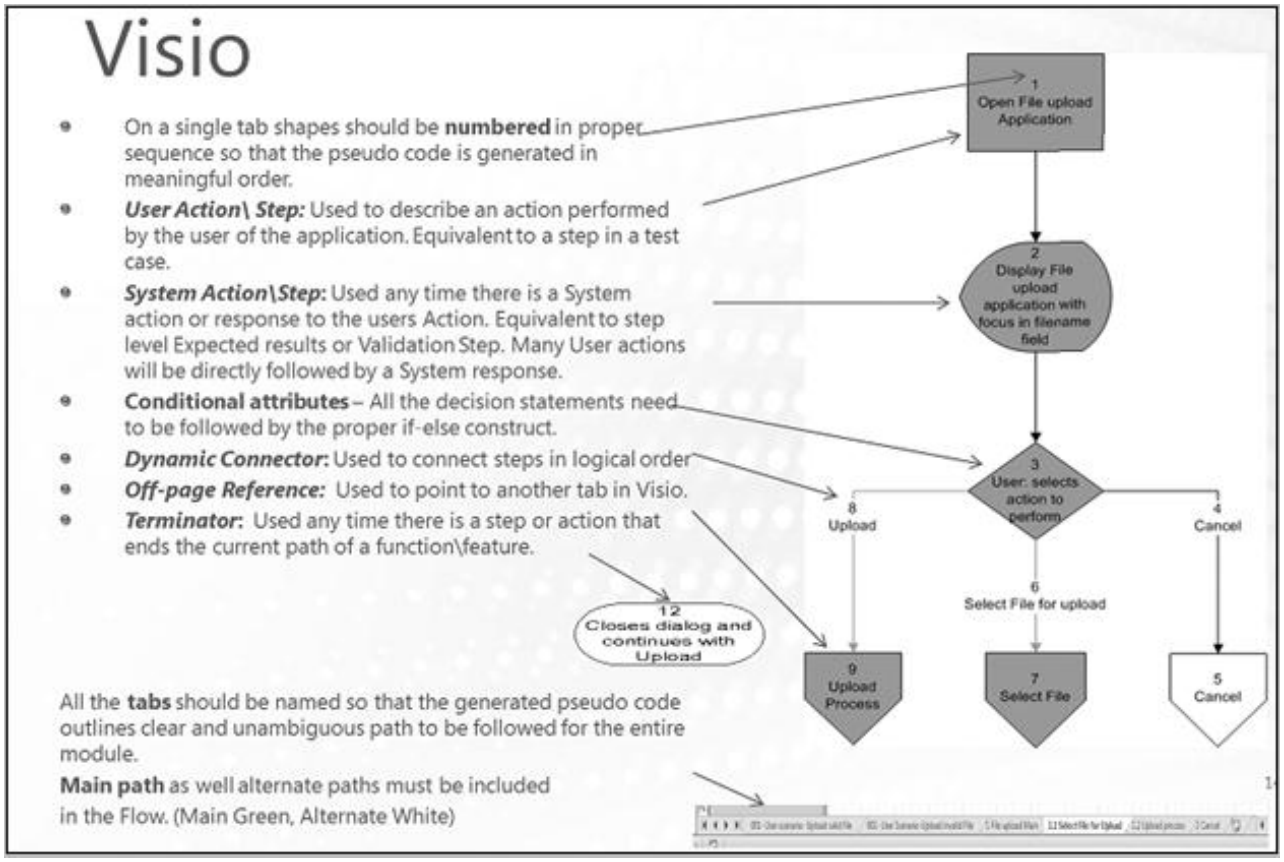


Figure 5: Shapes in Visio

Data Matrix: Using the visual format of the test cases makes it easy to identify where different data may be required in order to navigate through all of the possible workflows. We document these different data variations using XML.

Pseudo-code: Using built in functionality in Visio, we extract the data from the workflow into pseudo-code. The pseudo code reads as steps in a test case that we import in to our test case management tool. The pseudo-code also includes the embedded meta-data such as IF\Else hints or Go to statements.

C# Class: Our automation engineers translate the pseudo-code in to C#. Since the pseudo-code is an extraction of the workflow it contains all of the business logic from the workflow so a deep functional understanding of the application is not required to translate into C#.

Automation Framework: We execute the C# classes along with the data matrices in our automation framework. Since the classes are written in C# you could use a variety of different automation frameworks.

7. Conclusions:

By incorporating these changes in to our testing strategy we have ensured that our goal of meeting the business needs remains our top priority. We have adopted these processes on 2 project releases in my group so far. One project was released in to production in January and the other project releases in September of 2010.

7.1 Lessons for Others

1. Use User scenarios as a base for all testing

   Our user's satisfaction has been much higher on these projects than on projects in the past and this has increased their confidence on our ability to deliver to their expectations

2. Visually represent your test cases

   By presenting our test cases in Visio we made it much easier for other disciplines to review and provide feedback on test cases. In my ten years at Microsoft this has been the first project that we have had 100% review coverage of our test cases. This resulted in much higher quality test cases than we have seen in the past.

   Also due to the quality of the pseudo-code being generated we have been able to reduce the skill requirements of our automation engineers and have saved over 150k this year(This is based on the cost per hour difference between vendor levels multiplied by the number of hour spent on the project)

3. Move QA Process Upstream

   By involving our team earlier in the project we have been able to identify bugs before receiving an official drop from development. This has allowed us to test much deeper functionality that would have previously been blocked. On average we identified about 10 bugs per application build that fall in this category. So far on the project this has been a savings of 180k. (We spend about a thousand per bug fix and have 18 drops scheduled for the current project. This will be a reduction of over 180 bugs)

4. Extract data variations in to data Matrix

   We have also seen a dramatic cost savings in the areas of automation. By extracting our data matrix in to a separate entity we have reduced the number of hours required to maintain automated test cases by 4000 hours.(We reduced the number of test cases from 1048 to 128 and on average we spend about 4 hours per project per test case on maintenance)

8. Acknowledgments

For their valuable feedback and review:
Hisham Gaber
Yogesh Kulkarni

Driving Product Quality towards Release Goals

Bhushan Gupta
NIKE Inc.

Abstract

The software quality assurance activities strive to achieve utmost product quality at release given the scope, schedule, and resources. Normally, a program management team, comprising of all stakeholders, establishes a release criteria that functions as a measuring stick for the product quality at release. At a minimum, the release criteria includes test execution and defect goals from the quality perspective.

This paper describes a set of metrics that a software quality assurance group can provide to the program management team to assess the product quality and make release decisions. The author divides the quality activities during the product development into two phases; “New Functionality” testing and “Regression” testing and discusses test coverage such as total test cases planned, percentage of planned test cases executed and defect characteristics; such as number of defects by severity and their trends in each phase. As one would expect the defect find rate is a function of multiple factors including testing phase, test effectiveness, execution rate, and product complexity. Tracking each factor provides insight into on-going product quality and therefore an understanding of progress towards release. The author applies the concept of “code volatility” to further validate the defect characteristics. The trends in defect behaviors are explained with regards to the influencing factors highlighted above.

Biography

Bhushan Gupta has twenty five years of experience in software engineering, fifteen of which have been in the software industry. Bhushan recently joined the Global Store Solutions group at NIKE Inc. as a Software Quality Lead after a thirteen year tenure at Hewlett-Packard. He joined HP as a software quality engineer in 1997 where he was responsible for identifying key process areas to reduce rework. Based upon Hewlett-Packard quality parameters he developed quality goals, quality plans, and software validation strategies for several products. As a software process architect at HP, he customized the agile lifecycle for different groups and developed productivity metrics.

From 1995-97 Bhushan Gupta worked as a Systems Analyst at Consolidated Freightways. He was a faculty member of the Software Engineering Technology department at Oregon Institute of Technology from 1985 to 1995.

As a change agent, Bhushan volunteers his time and energy for organizations that promote software quality. He has been a Vice President, a Program Co-Chair, and a member of the board of directors of the Pacific Northwest Software Quality Conference.

Bhushan Gupta has a MS degree in Computer Science from New Mexico Institute of Mining and Technology, Socorro, New Mexico, 1985.

Introduction:

Data-driven decision-making takes out the gut-feels, hunches, and emotions from the decision making process and provides an analytical approach to make informed decisions. A critical decision an organization makes is when to release a product. Releasing an inferior quality product often has higher support cost. Delaying a release to achieve perfection incurs development cost and lost opportunities causing a revenue loss. It is only prudent to strike a balance between the support cost and the cost of delayed release. The balance is maintained by enforcing a release criteria once the business goals have been established. Once adopted by the stakeholders, the release criteria becomes the measuring stick and is a static control document throughout the product development.

Typical release criteria for a software product includes requirements coverage, defect find trend – normally declining for a certain period after the new functionality has been tested, and number of high severity defects that might adversely impact the support cost. The scope of this paper is the quality assurance metrics and actions that will drive the product quality towards a scheduled release.

Test Execution Phases:

To establish common understanding and clarity, it is necessary to understand the nature of activities involved in testing and how they can impact the quality metrics. In a broad sense, the testing lifecycle can be divided into two phases: testing new functionality and regression of the existing functionality. The two phases have very distinct activities as described below:

Testing New Functionality: This is the most prevalent reason for validating a new or a follow up release and has the following primary activities:

- Enhanced understanding of the product functionality
- Building test strategy and developing test cases
- Enhancing test cases for effectiveness and efficiency
- Clear and concise defect logging
- Helping developers with defect reproduction and further characterization

Regression Phase: In this phase the testing goal shifts to verifying defect fixes and determine any adverse effects caused by the fix on the already working functionality. The activities in this phase are:

- Re-execution of test cases to verify a defect fix
- If a fix fails, reopen the defect
- Re-execution of a subset of test cases to assure that a fix has not broken the existing functionality. Determination of the sub-set of test cases requires careful consideration.
- Finding new defects – the probability of finding new defects is low.

Release Criteria:

Formal or informal, each product development effort has a release criteria. A release criteria manifest is a list of conditions agreed upon by the program management team that should be met for market release. Its purpose is to achieve marketing goals with no or minimal support cost. Once established, it becomes a control document and guides the product development. The release criteria can only be altered if the business conditions have changed. Rothman [Rothman 2002] has discussed the mechanics of developing a release criteria and its use to determine the software release readiness.

The level of sophistication of a release criteria also depends upon the scope of the product being delivered; a hot fix, a patch, re-release, or a brand new product. At a minimum, a release criteria includes:

- Requirements coverage
- Percent test coverage
- Defect severity find rate declining over a period of time

A more sophisticated release criteria would have additional aspects such as:

- Maximum allowable number of defects that have been fixed but not verified
- Maximum allowable number of open defects by severity
- Declining code volatility trend
- Release Readiness
    - Training and support plans readiness
    - Completion of invention disclosures

The criteria may vary between organizations and within an organization. There may be other items in the release criteria if the software product has life threatening potential. Tabulated below is an example of a typical release criteria manifest:

Table 1: A Release Criteria Manifest Example

Criterion	Description	Owner
Functionality	100 Percent of the planned requirements completed	Development Manager
Test Coverage	100 percent test cases executed	QA Manager
Defects	• 100 percent "high severity" defects addressed as either: 1. Fixed, verified, and closed 2. Workarounds available where possible 3. Support cost estimated and agreed upon • 100 Percent customer impacting "medium severity" defects understood • 100 percent defect fixes verified • Find rate declining for 3 consecutive weeks • No more than 10 percent increase in overall number of open defects in the application since last release	Program Manager
Support Readiness	• Release notes up to date with workarounds where available • Documentation/Training material ready	Support Manager
Marketing Readiness	• Rollout plans ready and communicated	Marketing Manager
Development	• Intellectual property activities completed • Open defects moved to next release	Program Manager

The rest of the paper describes the metrics that can be used to track the quality of the product during product development to meet the release criteria.

Is testing on track? Requirements Coverage:

The requirements define product functionality which in turn establishes the product marketability. Once established, an organization's goal is to develop a product that meets all the planned requirements. It is imperative that the organization tracks the product requirements that have been fulfilled. For a quality assurance group it is important that all the requirements are validated. There is often a one-to-many mapping from the requirements to test cases. From the quality assurance perspective, the requirements coverage translates to test coverage and thus a QA organization measures the requirements in terms of test coverage. Quality assurance tool such as "Quality Center©" provides mechanisms to build traceability between requirements and test cases. In the absence of a tool, a QA team often uses a spreadsheet. The set of test cases are simply associated to a functionality and are tracked by it.

The requirements coverage is measured by the number of test cases that have been executed at any time during the product development. It is traditional to assess the development progress each week.

The QA also measures progress by the number of test cases executed each week. A program manager who is interested in a timely completion of the program will often ask QA 'Is testing on track?". A development manager will be interested in finding which functional aspects of the product are complete. The QA manager can address both of these questions using a simple metric shown in below:

Table 2a: A Simple Test Coverage Metric

Functional Area	Test Cases Planned	Planned for execution till date	Executed till date	% of Total planned Executed	% Test Cases Passed	Test cases blocked
User Interface	45	35	32	71	98	3
Database	195	45	12	27	75	8
Output Display	25	20	15	60	99	2
Reporting	40	10	10	25	50	1
Total	**305**	**110**	**69**	**22**	**87**	**14**

The metric emphasizes the following points:

- How is each functional area progressing towards release? The delta between planned for execution till date and executed till date is the measure of progress towards release. If the delta is positive or zero the test coverage is well on schedule.
- What is the quality of each component? Percent test cases passed signifies the quality of each component. The lower the rate of test cases passed, the lower the quality and attention is needed.
- What is the outlook for the future? The extent of blocked test case adversely impacts the future progress. A higher number of blocked test cases means more work is passed on to the future thereby back loading testing activity and potentially impacting the overall quality and the release schedule. In addition to assessing the quality of each functionality, the metric also provides a measure of overall progress and current quality of release.

Based upon the answers to these questions the organization can adjust resources in the test group or provide additional support to the development group.

Normally the data is collected at regular intervals and can be compiled as trend chart for each functional area. The overall result can be presented as a Status Dashboard for the program management and the development team to evaluate the release status as shown below. The data is derived from the test coverage metric (Table 2a).

Table 2b: Status Dashboard Showing Quality of Various Components of a Product

Functional Area	User Interface	Database	Output Display	Reporting
Status	Green	Red	Yellow	Red

The color notation has standard meaning and communicates the following:
Green – the quality is progressing as expected
Yellow – the product quality is not progressing as expected but the progress will improve
Red – The product quality is not progressing as expected and will require specific measures to improve

In the example above, both the Database and the Reporting functional areas are RED. For the database functionality only 27% of the planned test cases have been executed and the failure rate is relatively high (25%). In the case of Reporting, although the execution rate is 100%, the failure rate is high (50%). The two functional areas require immediate attention and a course correction.

Does the product have desired quality?

So far we have only considered test coverage as a component of quality. The other equally significant and might be even more critical component is the defects that have been discovered during testing. At any instant, we may have completed the planned testing but we are still finding a substantial number of defects. In addition, there may be an excessive number of defects that are yet to be resolved and still a significant number of defects fixed but yet to be verified to make sure that the fix is effective and has not caused any side effects.

The succeeding paragraphs describe some effective metrics used to address these concerns and make progress towards release:

Defect Find Rate:

The following graph illustrates a weekly defect find rate.

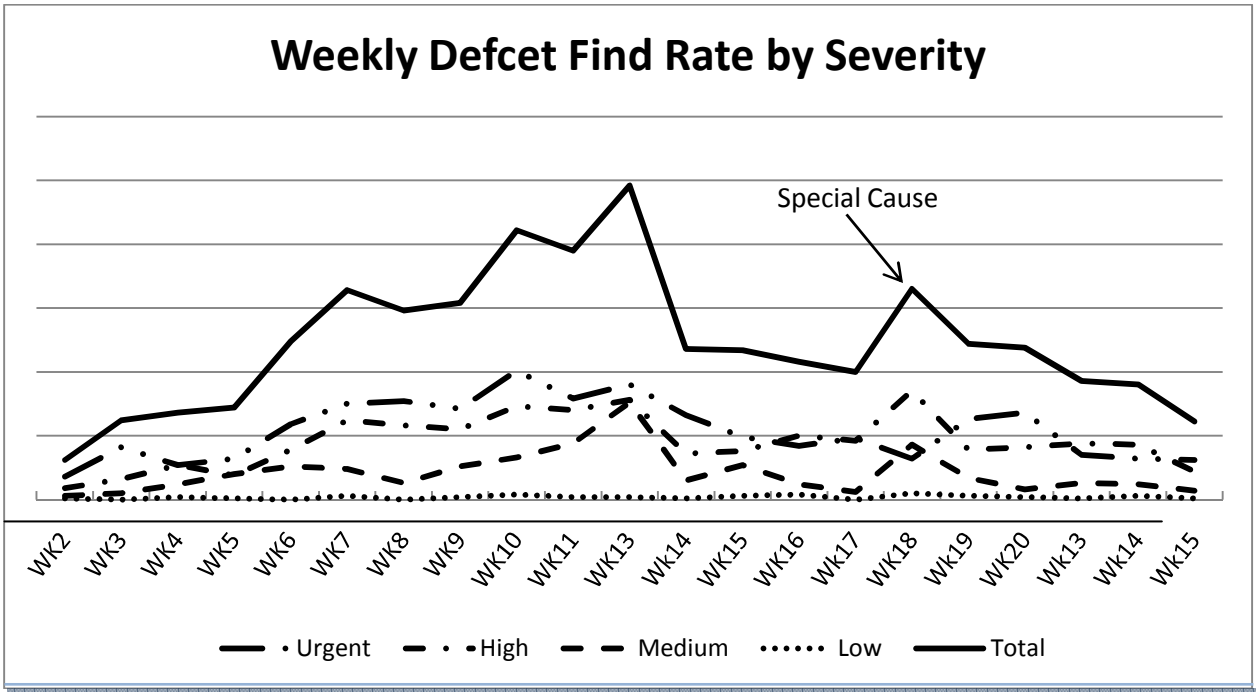


Figure 1 – Defect Find Rate by Severity

Initially, find rate is usally low due to:

- Limited understanding of the functionality by the test group on what the expected behavior is. This particularly is the case when the application workflow is complex or has been significantly changed in case of product versioning
- Planned functionality not yet complete
- Although the functionality is complete but not working end-to-end and thus certain aspects of the functionality are not reachable. This will be signaled by a high number of blocked test cases

As the testing progresses, the QA group better understands the functionality and the find rate increases. This rate peaks out when all the new functionality has been tested and the testing moves to regression. During the regression phase, only regression tests are run, fewer defects are found, and find rate starts to decline. As the development progresses, the find rate continues to taper off. It is not practically possible to find all the defects in the code so normally the release date sets the point in time when the testing should stop. Simmons [Simmons 2000] has modeled the defect arrival using the Weibull curve to answer the question “When Will We Be Done Testing?” Gupta [Gupta 2004] has utilized the concept of defect density to predict defect find rate.

While this is an ideal behavior there are normally fluctuations due to special causes such as test-blocking defects, extent of new code added during a typical period, and change in testing hours due to change in resources. Any steep fluctuation must be evaluated and a corrective action must be taken to address it as a special cause.

Find and Fix Rate:
Although the Find Rate plays a critical role in this metric, it only indicates the level of quality at a given time and fixing the defects actually enhances the quality. Of course, knowing the shortcoming of the product increases the chances of better risk mitigation. Fixing the defects is essential not to just enhance the quality but also reduce the support cost.

The following graph shows Fix Rate in combination with the Find Rate.

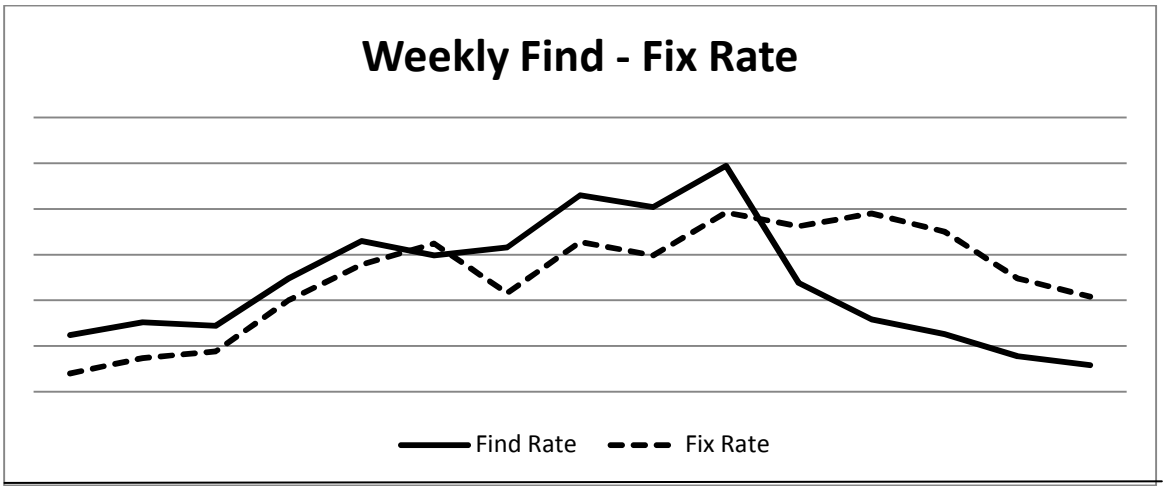


Figure 2: Find and Fixed Rates in a Test Cycle

In the beginning the Fix Rate trails the Find Rate. Once all the new functionality has been developed, the development efforts shifts towards defect-fixing resulting into a higher Fix Rate. At some point the Fix Rate surpasses the Find Rate. That is when one can mathematically compute the time required to fix all the known defects and the product quality at release.

Like the defect Find Rate, the defect Fixed Rate may be inflicted due to late feature creep resulting in new code. It is important that such behaviors are understood by the program management team.

Code Volatility aka Code Turmoil – the mother of all trends:

Any defect patterns that are used to make release decisions are inherently related to code volatility (sometimes also referred to as code turmoil). This is the number of lines of code that changes from one drop to another. After all, the amount of change in the code directly impacts the number of potential defects. The code volatility can be measured by accounting for new lines of code, modified lines of code and the deleted lines of code. A large number of deleted lines of code resulting due to removing a feature or re-structuring code has a small potential for introducing new defects. One needs to be diligent when inferring the number of deleted lines of code.

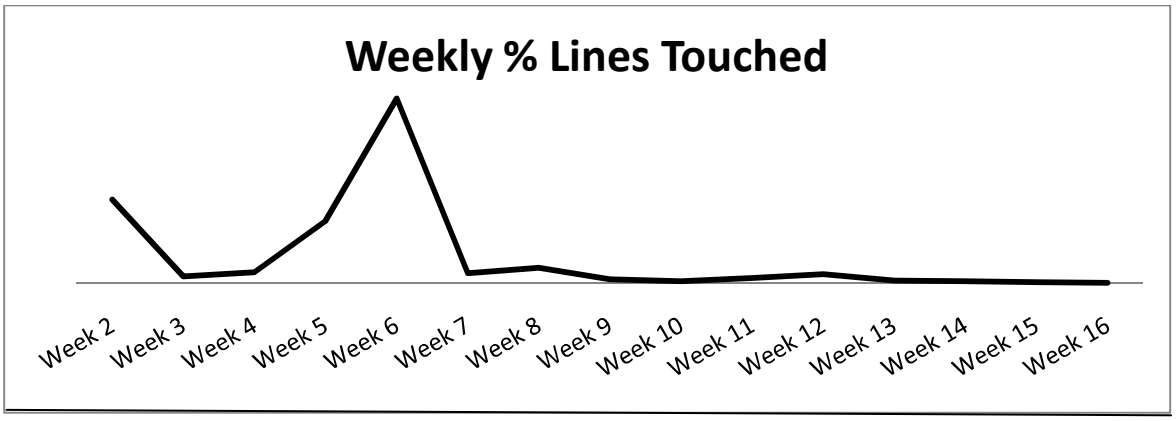


Figure 3: Code Volatility Trend in a Test Cycle

Once again, this trend provides a cross reference to the Defect Find Rate. As one would expect, the code volatility will be high during the development of the new functionality and will decrease as the development moves from the new functionality to the regression phase.

Although not commonly explored it is possible to correlate the number of defects to the code volatility as the development proceeds. Once established for a stable environment, it is entirely possible to predict the number of defects that are to be expected in the next code drop. This has the potential to evaluate the following facts while testing a new code drop:

- The number of defects anticipated
- The amount of efforts required to test
- Testing effectiveness measured by the number of anticipated versus actual defects

In some organizations a maximum allowable code volatility is imposed to keep the number of new defects to a manageable level. This becomes increasingly relevant as the release date approaches.

Defect Resolution for Scheduled Delivery and Quality

In order to release an intended quality product on time, it is important that the defects found in testing are resolved and verified in a timely manner. Defect resolution may not always require fixing a defect and a business reason should drive a fix. If a defect fix does not support a legitimate business reason, it does not have to be fixed at the cost of slipping schedule.

The defect aging metric helps provide an understanding of the defects that have not been worked upon for a longer than the expected period. Figure 4 shows how the defects with varying priority have been delayed for resolution.

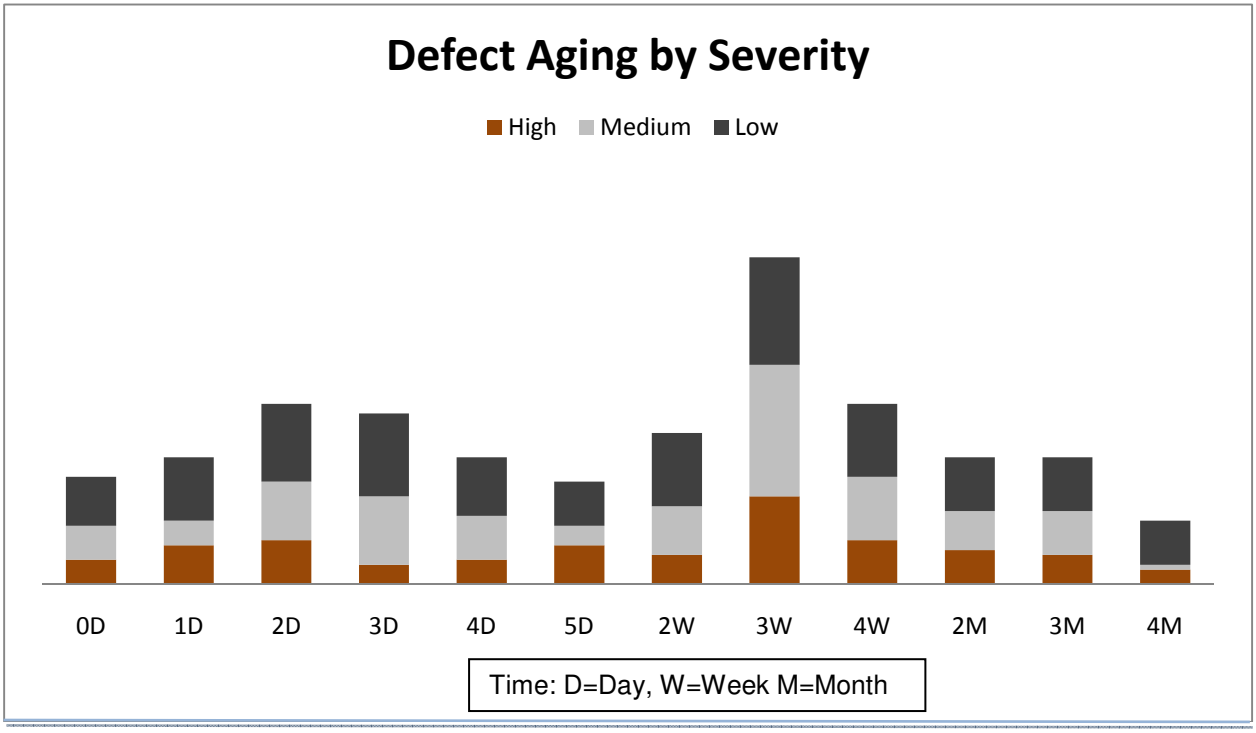


Figure 4: Open Defects Aging Chart

The graph shows how the open defects accumulate over time for each severity. In this example a significant number of defects are 3 weeks old. Letting the defects age makes it more difficult to fix as characterization becomes harder to reproduce. The defect verification activity also suffers with the same dilemma although not to the same extent since the test cases have been documented along with the expected results. An early decision on which defects should be fixed for the current release is beneficial to avoid spending longer than usual time on these defects.

Proactive QA Approach

The quote "Prevention is better than cure", applies well in the software quality arena. The QA group has an important role to play by means of work product reviews in the early phases of product development. Once the product has entered the validation phase the QA group can proactively facilitate a better quality product by taking some specific measures both prior to and after the product enters the development phase. Some of these measures include:

- Get familiarized with the product functionality – this can be achieved via reviews and working closely with the development teams on test development. Requesting an early drop of the code into QA is an excellent way to get familiar with the environment, installation and setup, and product workflow.
- Understand the high risk functional components from the development perspective and ensure that test case design and execution effort is appropriate to the risk.
- In the situation where test coverage is not on schedule understand the factors such as "test blocking defects" and "incomplete functionality" and coordinate that with the development group.
- Ensure that adequate resources are made available for testing. An understanding of testing capacity always helps to establish the need for testing resources.
- Provide as much assistant as possible to the development group with the defect root cause analysis by a clear defect description, providing screen captures and logs, and being available to answer questions.

- Apply alternatives to structured testing approaches when the test schedule is at risk. Testing is always on critical path and any proven test methods such as Exploratory Testing [Bach] will help the overall test schedule.
- Last, but not least, foster the "team approach with the development" and not "us vs. them".

Conclusion:

The goal of a quality organization is to support the delivery of a high quality product as defined in the release criteria. This can be better achieved by measuring the progress towards the goal along the product development. Gut feels and hunches do not lead to educated decision making and an organization using the right metrics will be in a better position to answer the question "Are We Ready To Release?". The author has extensively used these techniques and benefitted with their usage. A QA team should emphasize prevention over cure but cure is an important complement of prevention.

Acknowledgements:

The majority of the experience the author has shared in this paper comes from his thirteen year tenure at HP. The author has maintained a general approach and has especially refrained from sharing the data in its entirety.

References:

1. Rothman, Johanna; Release Criteria: Is this Software Done?, http://www.jrothman.com/Papers/releasecriteria.html
2. Simmons, Erik; When Will We Be Done Testing?; Software Defect Arrival Modeling Using the Weibull Distribution; 18th Annual Pacific Northwest Software Quality Conference, Portland, Oregon, October 2000
3. Gupta, Bhushan B.; Guiding Software Quality by Forecasting Defects using Defect Density; Pacific Northwest Software Quality Conference, Portland, Oregon, October 2004
4. Bach, James ; Exploratory Testing Explained, http://www.satisfice.com/articles/et-article.pdf

Working with Complex Applications

Engin Uzuncaova
Software Design Engineer in Test, Bing Maps
Microsoft Corporation
enginuz@microsoft.com

Abstract

Software testing, in general, is a human-centric process where creativity and technical skills mix and mingle in unique ways to both understand and evaluate software systems. When we are faced with new and original testing challenges, it is usually the human element in the process that makes it possible for us to repackage the existing tools and methods, again, in new and original ways for viable solutions. As the complexity of a system under test increases, this task becomes harder to accomplish. What do we do when the level of understanding required to properly analyze and test such systems is beyond our comfort zone?

Testing geospatial data and route planning algorithms is a good example for this; the inherent complexity of geospatial data, the representation of the data and also novel algorithms that consume this data make testing more of a daunting (or fun?) task at both levels: understanding and analyzing. This paper presents a testing approach that we have developed at ***Bing Maps***[1] to attack the complexity of our route planning service focusing on the following areas:

- Heuristic test oracles,
- Efficient test infrastructure,
- Rich visualization solutions for complex geospatial entities and scenarios, and finally
- Integration of statistical methods with the testing process.

This approach has provided for improved coverage in our testing and enabled us to identify more quality issues during development. We also benefited from a more efficient evaluation mechanism for complex issues found during testing. While our improvements in each individual area have proved to be highly valuable, the main benefit we observed with this approach has come from using all these improvements in tandem representing a strong test strategy.

Biography

Engin Uzuncaova is software development engineer in test for Bing Maps Directions team at Microsoft. He joined Bing Maps in October 2008 and he's been working on routing algorithms and data.

Previously, Engin interned at IBM working with the WebSphere Application Server team and at Google working with the Google Updater team. He received a master's degree in Computer Engineering from U.S. Naval Postgraduate School in 2003 and a Ph.D. degree in Software Engineering from the University of Texas at Austin in 2008; his dissertation was titled "Efficient Specification-based Testing Using Incremental Techniques".

[1] ***Bing®*** is a registered trademark of Microsoft Corporation.

1. Introduction

As software systems grow in size and complexity, testing them becomes more challenging, especially, since testing is often done manually. A key challenge is working with complex applications; i.e. applications that are based on sophisticated algorithms that use large and complex data structures. As we work with such applications, it becomes tougher for testing to achieve higher confidence and greater efficiency.

Testing starts by understanding the problem and then the solution, i.e., the software under test (SUT). As our understanding gets better, we develop more effective testing solutions to help us achieve the desired level of quality. Based on my experience with the route planning service in Bing Maps for almost two years, I believe it is a difficult testing problem. This paper presents a testing approach developed at Bing Maps to tackle the difficult challenges of working with complex geospatial data and sophisticated route planning algorithms.

Route planning service, in the simplest terms, can be considered as a user-friendly solution to the well-known shortest path problem [Cormen09] that is defined for an actual road graph. For today's mapping services, these graphs tend to include millions of edges and the algorithms that are used on this data are bounded by many functional and non-functional requirements.

When you are testing the route planning service, you mostly work with geospatial objects that are represented, basically, with coordinates (i.e., floating point numbers). Putting this into a context, given a user query (such as the driving directions from point A to point B), a routing algorithm finds an optimal path by examining the gigantic road graph. There are plenty of different routing algorithms that can be optimized based on various quality criteria [Delling09]. In the road graph, each edge represents an actual road segment with a large set of primitive properties such as speed and length; and some complex properties such as turn restrictions.

This paper focuses on three main problem areas that we have constantly dealt with in this domain:

1. The route planning domain involves many interesting and complex scenarios that need to be rigorously tested such as dense highway systems and missing/incorrect road data. Defining tests for such cases can be a tedious task. As the routing data gets updated, tests may become obsolete, which we may need to replace with valid ones.

2. Testing the route planning service requires executing the tests on a large set of test inputs, which usually goes through multiple iterations. Any inefficiency in test infrastructure causes unproductive test development cycles and lengthy execution and analysis times.

3. The main hurdle with test results in this domain is that there are usually large test suites used for certain functionalities and, as a result, there is a large body of results that need to be analyzed. This can be a difficult task when results represent complex test criteria, especially when optimizing subjective areas such as route-quality.

2. Example: Evaluating a Behavioral Change in the Routing Algorithm

This section illustrates a specific testing problem for evaluating a behavioral change in the routing algorithm and presents an approach to address the issue.

2.1. The Customer is Always Right..!

In a typical product life-cycle customer feedback is a constant input to quality improvement process. As an example, let's assume that we have received significant amount of customer feedback indicating that our driving directions favor low-quality roads, hence suggesting longer and non-ideal driving directions. After an investigation, the feature team verifies the problem and designs a solution to mitigate the route quality issue. Figure 1, hypothetically, depicts the original behavior on the left preferring inner roads, and the new behavior, on the right, preferring highways. It is now in the hands of the test team to check that the original issue is resolved, and also there is no undesired impact on the rest of the behavior.

2.2. Testing Cannot Show Absence of Bugs

With the new design, we can safely assume that some driving directions will be different than before, hopefully in a positive way. However, keep in mind that this is a subjective change, which cannot be measured with the same precision as a litmus test; making human judgment an important part of the evaluation process. In addition, we might need to go through multiple test iterations to optimize the new behavior and repeat the tests for different configurations, which necessitate a reasonably quick turnaround for test passes. Moreover, we might need to check a number of relevant non-functional requirements depending on the nature of changes introduced by the new design such as performance.

One of the main steps in this process is test generation, i.e. constructing a set of test inputs to be used for assessing correctness and quality. The test inputs we select must provide adequate coverage (e.g. geographical, functional and code). Obviously, given the size and complexity of the routing data, exhaustive testing would not be feasible and analyzing the results of a large test suite would introduce a substantial overhead. If we don't have reusable tools in place, developing an automated solution to do the trick might turn out to be a costly investment for such a specific problem.

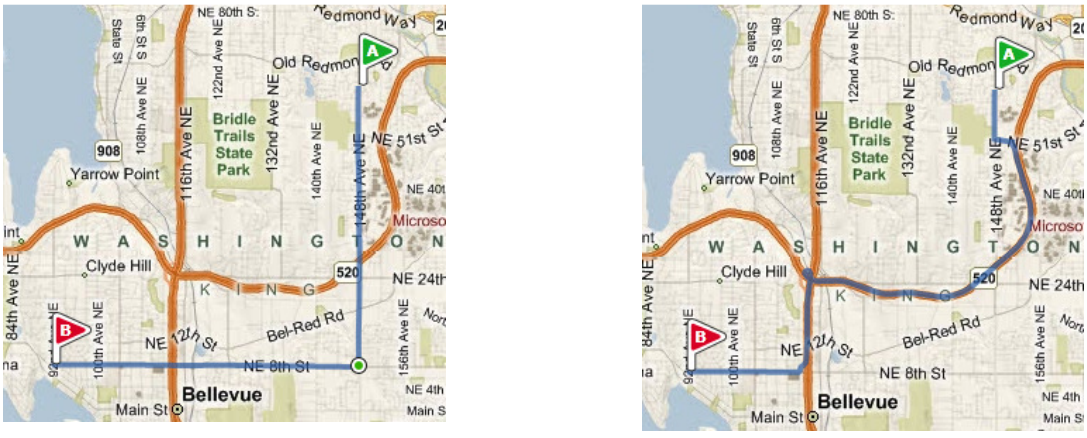


Figure 1 - Two different driving directions showing the original behavior on the left and the new behavior on the right.

As much as we would desire a minimal test suite, it is not unusual to work with thousands of test inputs in the routing domain for behavioral analysis. For instance, this is true for geographical coverage; coverage can only improve with more tests. As we increase the size of the test suite, however, test execution becomes more of a bottleneck considering that we will need to run these tests multiple times for different configurations.

Adding on top of that the ever-increasing pressure to release new features (to the feature-hungry end users), there is reasonable motivation to be traditional and stick to what we have at hand, let it be manual or automated. This pressure also leaves little room for a full-fledged formal analysis of the change.

2.3. An Approach to Tackle the Problem

For this particular problem, we used a number of different approaches and tools that we have invested on for some time. Working on similar problems over a prolonged time has given us the opportunity to think about various key investment areas to increase productivity and efficiency of our test efforts.

2.3.1. Test Execution

With a large enough test suite, it is more likely to achieve a higher confidence in our test results; however, this is generally restricted by the amount of time that we can afford. To mitigate this limitation, we redesigned our test framework to support parallel test execution. With this capability, we achieved an order of magnitude improvement in test execution time. This translates into being able to run tests faster, hence enabling shorter turn-around period for test feedback. It has also allowed for scaling up to larger test suites more efficiently.

2.3.2. Test Generation

One of the main issues with this kind of a behavioral change is measuring the unintended results of the change (i.e. regression). For that, we need to achieve significant coverage for topological entities (such as streets, state-highways, interstate-highways and their intersections) and geographical regions (different parts of the world represent dramatically different road structures).

One of the direct ways to generate test suites is to use guided-random test generation to improve coverage based on certain criteria. For example, a rich set of test inputs can be generated using custom graph traversal algorithms that run on top of the underlying routing data.

2.3.3. Heuristic Test Oracles

Heuristics provide an approximation to the optimal solution, which is particularly useful when what is correct cannot be defined precisely. As we get better heuristics, the chance of identifying issues with heuristics also increases. For our scenario, numerous different heuristics can be used to identify potential issues ranging from simple ones such as differences in driving distance and duration, and the number of turn instructions issued per route to more complex ones such as ranking driving directions on a set of predefined criteria to measure quality of the generated results.

2.3.4. Visualization Tools

Geospatial entities are usually represented complex properties and coordinates. Within the context of geospatial analysis, working with this data in the raw format is neither intuitive nor easily comprehensible. For example, let's assume that we are trying to understand why and how two driving directions are different. Key elements of this analysis are the properties of the geospatial entities (i.e. roads) and the algorithm behavior (i.e. optimum path selection). Using a very primitive approach as described above and looking at the raw data next to each other is, in its simplest terms, painful (imagine that doing this for hundreds of different instances!). Instead, a visualization tool that presents the essential data pertaining to the analysis in an intuitive way can play a vital role in the assessment process.

3. Problem Areas

This section describes the problems we have experienced while testing the route planning service. We categorized the challenges in this domain into three areas as in the following sections.

3.1. Test Generation

One of the weakest areas of testing practice is test input selection/generation. A common pitfall is the ***hello-world syndrome***, as I call it. This can be described as the tendency of testers to use the most basic instances as test inputs especially when the test inputs are complex data structures. The main reasons for this include but are not limited to the lack of in-depth understanding of the SUT and the overhead of generating a large set of test inputs. In the long run, however, this pattern of behavior causes some gaps in test coverage, which triggers significant costs later on in the development process.

The desired approach in test generation is to test the SUT for all possible combinations of test inputs; however, this is usually infeasible, if not intractable. Therefore various techniques are used to sample the test input domain to satisfy certain coverage criteria such as equivalence partitioning and boundary value analysis [Hamlet88]. What makes such techniques less useful for complex data structures is because of the mere fact that they are complex and maintaining a consistent test suite can be a challenging task if the underlying data and/or its representation is dynamic in nature (such as changes in the routing). For example, a valid test case for the right turn instruction based on a boundary test case can become obsolete if the routing data updates the topology at the intersection even slightly.

To illustrate, think about the problem laid out in the example in Section 2. The optimization aims at improving the quality of the directions generated by the algorithm. As the algorithm is modified, the results that it generates are likely to be different (in the case that they are not different it is another interesting problem because our intent is to alter the behavior in a certain way). There are different coverage criteria when selecting the test inputs such as structural properties of the routing graph, geographical coverage and code coverage. To satisfy these criteria, manual approach is not an option due to being a tedious and error-prone process, whereas automated methods can prove to be useful in scaling up to generating large test suites.

3.2. Test Execution

As mentioned before, the route planning service works on top of a very large data structure that represents the road graph. The amount of functional test points and the corresponding coverage criteria

for each of these points necessitate use of large test suites. In addition, these tests are usually run multiple times in conjunction with the development efforts to evaluate certain aspects of the underlying data and the algorithms iteratively. Changing just one-line of code in the algorithm can initiate a lengthy test run including thousands of test cases. Automation is the preferred solution for executing these tests due to obvious efficiency concerns compared to manual approach. Even then, test execution can be a bottle neck especially under time constraints, which we are all very familiar with.

Continuing on the same example, executing the selected test suite involves generating a route result and collecting the required information for evaluation. Depending on the selected test suite, the execution time can be significant and causes longer turn-around time for test feedback and scheduling conflicts among other tests that may require the same resources.

3.3. Test Evaluation

Working with complex data introduces challenges not only for test generation and execution but also for evaluation and sometimes in a much more unanticipated ways. The evaluation process is usually full of false-negatives and false-positives. For example, a route-quality optimization may work for longer driving directions but for very short ones it may introduce an undesired behavior. In some other situations, data and functional issues may be hard to distinguish from each other; for example, what is reported as a non-optimal result may in fact be due to a problem in the routing graph such as incorrect road-speed or turn-restriction. Moreover, sometimes test results may fail to detect subtle issues, such as increased use of left turns due an optimization in the algorithm. Issues like this require a deep understanding of the subtleties in the domain, which can be gained through experience. The crucial component of an effective evaluation process is presentation of the relevant test data in intuitive ways.

Think about route results in general; they have a set of edges that represents a shortest path in the routing graph. Each edge has many properties such as connectivity, speed, shape, restrictions and many others. How do we go about verifying that the generated path is in fact the shortest? Besides, a result can fail in many different ways; a turn restriction may be violated, a non-optimal result can be generated or driving instructions may be incorrect. Each combination of these factors, as a failure, requires human involvement to verify the issue. Collecting all required data points on a case by case basis through manual means can bring evaluation process down to highly inefficient and unproductive levels.

4. Our Approach

This section describes our approach to address these issues presented in Section 3. We invested on four different areas: (1) improving the test infrastructure, (2) using heuristic-based testing, (3) using statistical methods, and (4) developing visualizations tools.

4.1. Improving the Test Infrastructure

The main motivation for improving the test infrastructure is to be able to transition into more productive test development process and be more efficient in test execution (Section 3.1).

4.1.1. Parallel Computing

To achieve a certain level of geographical coverage, we usually need to run our tests on a large number of test inputs and, especially for algorithm optimizations, usually through multiple iterations, which can end up taking significant amount of time. One obvious way to address this scalability issue is that we can reduce the number of test inputs so the tests complete faster. However, this approach has a direct negative impact on test coverage. Instead, we can explore opportunities that lie in the parallel computing area. Today, even an average desktop computer is configured with multi-core processors. Most of the time, tests are sequential and don't depend on each other, which makes it possible to run them in parallel. This is very simple to achieve and delivers a dramatic increase in efficiency in test execution time. We have seen an order of magnitude speed-up using this approach.

4.1.2. Developing Common Libraries

It is an unfortunate fact that better software engineering practices are not usually followed in the testing domain. With constant release pressure and our hard-to-change tendencies based on waterfall development processes, it is really rare to see testers improving the quality of their own code. While this faulty behavior is generally justified by the amount of time saved for actual testing, ignoring this area almost always introduces adverse effects in the long run.

By recognizing this, we invested on developing common libraries for test framework covering the following areas:

- Test configuration
- Test generation
- Service proxies
- Database-specific operations

Among many, the main benefit of having common libraries for testing is that we are now able to develop test programs in a much productive way since most of the complex operations are encapsulated in these libraries and we can better focus on developing actual tests for critical scenarios.

4.2. Using Heuristic Test Oracles

Using heuristic test oracles for testing provides an effective approach for testing, especially when the correctness criteria for a test cannot be defined precisely [Douglas99]. In the route planning service we have many such scenarios; the following section lists some of these scenarios and the heuristics that we have used extensively in our testing.

4.2.1. Checking Driving Instructions

The route planning service, in general, generates a shortest path and a sequence of instructions to help navigate along this path such as turn instructions. Road topology and other properties are commonly used to control the level of detail of the instructions shown along the path. A simple test to ensure that the instructions are correct requires a complete test oracle, which is essentially a redundant implementation of the SUT. Instead of using such a complete oracle, we can use the following heuristic to help us identify the potential issues:

- Identify the direction of a given instruction (rather than the instruction itself) based on the angle between road segments involved.
- Evaluate sub-paths to identify cases where too many instructions are presented in very short distances.

4.2.2. Checking Algorithm Behavior

For brevity of the discussion, let's assume that we employ a shortest path algorithm that runs on the road graph and generates a shortest path for any given pair of start and end locations. Similar to the discussion in the previous example, designing an oracle that returns the correct answer for each route query is impractical. With the following heuristics, we can easily achieve significant test coverage:

- The length of a shortest path cannot be smaller than the great-circle distance [Wiki_01] between the start and end locations.
- Any sub query along a parent shortest path must return a shortest path that is a subset of the parent. Using this heuristic, we can test a route query iteratively until the start and end locations are the same.
- A shortest path that is more than the great-circle distance by a certain margin can be considered as a potential issue especially in densely populated areas.

4.3. Using Statistical Methods

Statistics is generally used to draw conclusions from data [Montgomery07]. Given that, it is hard to miss the fact that there is a direct relation between testing and the main premise of statistics, especially when we are testing complex applications. Considering the route planning service, there are two factors that make application of statistical methods more pertinent to our problem areas: Using large test suites and analyzing large bodies test results.

As described before, by using large test suites we intend to achieve better coverage. In order to accomplish this, the test suites that we use must represent correlation to effective coverage criteria. For example, if you are measuring performance, your test suite must correlate to the actual operational profile of you product. Similarly, if you are aiming at geographical coverage with your tests then your test suite must correlate to the actual coverage area of your service. The statistical methods become important when the coverage area is large (such as the entire set of user requests or entire service coverage area). Statistical methods, such as sampling, can be used effectively in these scenarios for test input generation especially in larger scales.

Besides test generation, the task of analyzing test results also represents opportunities for statistical methods. For example, confidence intervals can be used to assess significant deviations in data, which are likely to point to issues. Think about a scenario where you test a new routing data release. Out of 5000 routes that are calculated, 20% shows longer overall trip time. Looking at the results more closely reveals that only 0.1% of these results (actually five routes) have a difference of more than 10%. With careful use of statistics we can support our findings and eliminate noise in the results.

4.4. Developing Visualizations Tools

The previous chapters aimed at highlighting the difficulties involved in working with geospatial entities and presented some ideas for automated testing. Interestingly, visualization tools can be considered on the

other end of the spectrum emphasizing careful evaluation by skilled testers. We believe that these two approaches are not contradicting, but rather complementing each other. During testing, we work with and generate large amounts of data in the forms of test scenarios, test inputs and test results. In its raw format, this data is represented as geospatial entities with complex properties, which makes understanding and analyzing more of a tedious, difficult and error-prone task. For example, when the route planning algorithm generates a route result that does not look correct, it may be due to many different factors:

- Road data may have some problems such as invalid turn restrictions and incorrect speed,
- Algorithm may have some problems such as logical errors and incorrect optimizations,
- Data and algorithm may be correct but the scenario may represent an edge case, or
- Test configuration may be incorrect; hence, the result may be a false-positive.

As a tester, as much as it is challenging to be able to ask these questions while investigating the issue, it can be more challenging to get the answers if we don't have the proper tools available. Visualization tools help us comprehend the complex data and scenarios by presenting a rich set of relevant data in an intuitive way and improve productivity immeasurably.

To illustrate let's look at a simple case where we compare route results that are calculated based on two different routing data releases. As mentioned before, we apply some statistical methods and identify that a handful of results are actually of any significance. Using the available visualization tools, the results would be presented separately as it is shown in Figure 2 (a) and (b). With this approach, it is difficult to pinpoint how the two results differ. However, we can programmatically analyze these results and identify their differences, which then can be used by a visualization tool to provide a more intuitive depiction. Figure 2 (c) illustrates this approach.

(a) First alternative (b) Second alternative (c) Results are shown together.

Figure 2 - Intuitive visualization helps depict the issue more clearly.

5. Conclusions

Testing complex applications can be a challenging task for test generation, execution and evaluation. Given the unique challenges imposed by such applications, testing may become more difficult than development at certain times. Understanding the application domain and the common problems can present opportunities to develop effective solutions for testing to boost efficiency and productivity. The benefits of such solutions can be dramatic:

- Given the constant time pressure and the need for delivering novel and compelling features, test feedback plays a vital role in developing high-quality software; any improvement in test feedback quality and time is quite valuable for the entire team.
- Complex data introduces challenges to grasp the subtleties of the application domain; investing in new and original approaches such as heuristic test oracles can extend our capabilities to both understand and address the issues that may be otherwise hard to find.
- Rich visualization tools provide for increased productivity in assessing the issues at hand; smart investment in this area has a great potential for return on investment.

In testing, we are never short of new and original problems; as we gain more experience and maturity in our expertise, we can develop more effective solutions by focusing on the emerging patterns in those new problem domains.

References

[**Cormen09**] Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to algorithms*. The MIT Press.

[**Delling09**] Delling, D., Sanders, P., Schultes, D., and Wagner, D. (2009). *Engineering route planning Algorithms*. In Robust and Online Large-Scale Optimization (pp. 182-206). Springer Berlin / Heidelberg.

[**Hamlet88**] Hamlet, D., and Taylor, R. (1990). *Partition testing does not inspire confidence*. IEEE Transactions on Software Engineering , 1402-1411.

[**Douglas99**] Hoffman, D. (1999, March/April). *Heuristic Test Oracles*. Software Testing and Quality Engineering Magazine , 1 (2).

[**Montgomery07**] Montgomery, D. C., & Runger, G. C. (2007). *Applied Statistics and Probability for Engineers* (4 ed.). Wiley.

[**Wiki_01**] *Great-circle Distance*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Great-circle_distance.

Turning Complexity into Simplicity

Jon Bach

jonb@quardev.com

Twitter: @jbtestpilot

Abstract

This paper is an informal study in recognizing complexity and overcoming it to solve a testing problem. It enlisted the support of the Weekend Testers, a group of passionate professional testers in India who invite testers from all over the world to collaborate in weekend testing exercises. They asked me to be the host of one of their sessions, and this paper is about what happened when I gave them something complex to test.

Biography

At Quardev, Jon Bach is lead consultant and Manager for Corporate Intellect. He manages testing projects ranging from a few days to several months using Rapid Testing techniques like Session-Based Test Management – a way to manage and measure exploratory testing – which he and his brother James invented. In his 15-year career in testing and Quality Assistance, Jon has led projects for many corporations, including Microsoft, HP, Rational, and LexisNexis. He has presented keynotes and testing topics at over 60 testing conferences; written for testing magazines, and teaches testing techniques at Quardev and abroad.

Experience Report

Webster's definition of complexity seems like it has a sense of humor. They say it's "something complex" or "the quality or state of being complex."

That encourages one to look up "complex" which is more useful:

"Hard to separate, analyze or solve."

Sometimes complexity feels like trying to understand the cockpit of a 737. I talked to a pilot of such an airplane who, when asked how he deals with its complexity, told me: "I don't see it as complex. I only need a few of those switches and gauges at a time to do any one thing."

The secret seems to be that the pilot has a way of breaking down all of that stimulus, ignoring what he doesn't need and focusing on what he does need. While the definition of complexity says this is hard to do, I wanted to study how it could be made easy. (For example, a pilot's procedures and checklists are one way to battle the complexity of aircraft systems that need to work together to get people from one place to another.)

To study this juxtaposition further, I enlisted the Weekend Tester (India) crew to have them test what I thought was a complex application: Mifos, a microfinancing application by the Grameen Foundation. I chose this because I knew someone who worked for Grameen, who had asked me to give my impression of Mifos months ago.

I said it was one of the most complex things I had ever tried to model.

My charter was "Create a list of features toward creating a test coverage outline and for planning charters to be posted on the site for volunteer testers can run. Find every feature or enhancement you can find and be ready to discuss your questions and issues with project stakeholders."

I found over 250 testable features:

Feature Outline:

http://test.mifos.org:8085/mifos/AdminAction.do?method=load

Photo
Loan links
Add a note
See all notes
Edit membership
Office
- Add
- Define
- View
- Manage
- Edit

System users
- View
- Define new
- Manage roles and permissions

Perf history
- Number of clients
- Amount of last group loan if applicable
- Average loan size for individual members
- Total outstanding loan portfolio
- Portfolio at risk
- Total Savings
- Loan Cycle number
- Amount of last loan

- Total # of active loans
- Delinquent Portfolio

Schedule meeting

Roles
- Add new role
- Admin
- Loan Officer
- Manager
- Teller

Organizations
- View fees
- Define new fees
- View funds
- Define new fund
- View Checklists
- Define new checklist

Products rules and attributes
- View product categories
- Define new category
- View lateness/dormancy definition

Change Log

Apply Adjustment

Manage Loan Products
- View loan products
- Define new loan product

Manage Savings Products
- View savings products
- Define new savings product

Fees
- Fee calculation (mathematics / algorithms)
- Product fees
    - EF
    - ELDF
- Client fees
    - Application
    - Admin
    - Processing

View Office Hierarchy

Manage clients
- Create new group
- Create new member

Manage accounts
- Open new loan account
- Open new savings account
- Enter collection sheet data

Client accounts and Tasks
- Manage collection sheets
- Enter collection sheet data
- Print collection sheets

Create new clients
- Create new center
- Create new group
- Create new member

Create new accounts
- Create savings account
- Create loan account
- Create margin money account

Manage account status
- Change account status
    - Partial Application
    - Pending Approval
    - Active / Approved
    - Cancel
    - On Hold
    - Close
    - Blacklisted

Search

Your Settings

- Edit Info
- Change password

Login / Logout

Reports

- Client Detail
- Center
- Status
- Misc
- Performance
- Loan Product Detail
- Analysis
- Branch Office (dropdown list)
- Loan Officer (dropdown list)
- Loan Product (dropdown list)

Print Collection Sheets

Administrative tasks

Manage organization

- System users
    - View system users
    - Define new system user
    - Manage roles and permissions

Offices

- View offices
- Define a new office
- View office hierarchy

Organization Preferences

- View fees
- Define new fees
- View funds
- Define new fund
- View checklists
- Define new checklist
- View holidays
- Define new holidays
- Define accepted payment types
- View PPI settings
- Configure PPI Settings

Data display and rules

- Define Labels
- Define Lookup Options
- Define mandatory/hidden fields
- View Additional Fields
- Define additional fields

System Information

- View System Information

Manage products

- Product rules & attributes
- View product categories
- Define new category
- View lateness/dormancy definition

Manage product mix

- View products mix
- Define products mix

Manage Loan products

- View Loan products
- Define new Loan product

Manage Savings products

- View Savings products
- Define new Savings product

Manage accounts

- Manage Loan accounts
    - Reverse Loan disbursal
    - Redo Loan disbursal

Manage reports

- View admin documents
- Upload admin documents

View reports templates
Upload report template
View reports category
Define new report category
Manage surveys
View surveys
Define new survey
View question bank
Define questions

* went here to find new features:
http://www.mifos.org/developers/testing/1.1-test-strategy#objective-1

Adding and Removing Group Membership
Administrative Documents
Configuration
Group Loans with Individual Monitoring
Holiday Handling
Limiting Product Mix
Loan Defaults
Loan Schedule Independent of Meeting Schedule
Multiple Adjustments
Redo Loan Disbursal
Surveys
Undo Loan Disbursal
Help

Went to Roadmap to see other features:
http://wiki.java.net/bin/view/Javatools/RoadMap
BIRT reporting system
CGAP reports
PPI
Increased Support for Individual Clients/Teller Model:
- Ability to add group membership and to remove group membership to/from individual clients
- Ability to disburse loans on non-meeting days
- Ability to schedule repayments on non-meeting days
- Receipt and voucher printing from account pages
Accounting & adjustment enhancements:
- Undo loan disbursal / Redo loan disbursal
- Ability to adjust multiple loan adjustments
- Off-setting
Product Definition Flexibility:
- Loan defaults based on previous loan amount or loan cycle
- Calculating mandatory savings deposit amount on outstanding loans size
Moratorium Requirements
- Bulk loan creation
- Ability to restrict loan product mix
- French Localization
Enhancements
Reduce runtime of unit test suite to < 10 minutes
Improve handling of localized text
Upgrade Hibernate
Fix Look-up Value Overwriting
Support for additional lending models:
- Joint liability group (members of a group are held accountant able for loan repayments of others in their group; rules can vary, but examples include: group members can't receive a new loan a member of their group has a loan in arrears;)
- MGG
New Products:
- Insurance
- Shares
- Term Deposits
Increased Loan Product Flexibility:
- Additional repayment options: daily, flexible definition (ie, able to define outstanding amounts by month), able to edit specific repayment amounts
- Support for multiple (trache) disbursements for same loan, balloon repayments, etc
Enhanced Reporting Capability
Accounting Tools:
- Robust loan rescheduling

- Collection against write-offs

Adjustment Tools:
- Ability to adjust a single historical loan payment
- Ability to adjust single historical savings deposit/withdrawal/interest

Data Migration Tools:
- Tools for manual data entry of historical data
- XML (or another format) support for automated data migration

Accounting Interface: via batch file

Offline support: for Loan Officer daily tasks

Archive support: Ability to define rules for trimming database and summarizing data (ie, after N years, save only year end balances for savings accounts and archive savings transactions).

Branch level holidays

Support for additional lending models:
-SHG/Sacco support that tracks individual repayments separately from group repayments
- Full teller-model support: Removing meeting requirement in Mifos

Multicurrency Support?

Enhanced MFI Configuration Settings:
- Data scope configuration (can a branch see data from another branch)
- Additional levels of office hierarchy
- Defining lending models by branch

Enhanced Product Configuration Settings:
- Product availability by branch
- Products by lending model

Enhanced Loan Functionality & Flexibility:
- Automatic Calculation of Penalties
- Collateral Tracking
- Business Performance Tracking (can be handled via surveys?)
- Changing way loan cycles are handled
- Linkages to savings and/or Shares: Balance and Ownership requirements
- Configurable rules around early loan repayment
- Interest due calculation based on actual payments
- Payment via account transfers

Enhanced Savings Product Functionality:
- Savings account fees
- Savings acct restrictions (min balance, min amount to receive interest, max withdrawal amount, max # of withdrawals, etc)

Enhanced Client Data Collection:
- Biometrics collection
- Clients can belong to multiple groups/centers
- Codification of village/city/towns

Enhanced Work Flow & Permissioning System:
- Configurable Work flow management tools
- Field Level Permissioning
- Ability to combine loan and client/group approval steps into single step, ie, create client/group/center/loan all at same time

Interfaces:
- ATM integration
- Cash Management Systems
- Front-end POS devices
- Payment Systems
- Regulatory Agencies
- Kiosks
- Mix Market
- Smart Cards

Cash Management Support:
- Includes things like track clearing and tracking, deposit tracking, bank account balance tracking and management of multiple accounts, etc
- Tracking fund balances

Securitization Support:
- Portfolio tagging at account level
- Report generation

Misc:
- Easy tools for "Splitting" a Branch
- Ability to group clients into "Programs" (ie, HIV program, beggars program, etc)
- Set-up Wizard
- Localization into additional languages
- Self-extracting Windows Installer
- Collection of structured data in "Additional Data Fields"
- Support for variable/floating interest
- "Cash taken to Field" added to bulk-entry

The magnitude of this application stuck with me for months. I could take any one of those features and ask handfuls of questions about it, building thousands of tests. I could take one line in this outline like "Support for variable/floating interest" and have meeting after meeting on what it means and what it could mean. I could use 36 different test considerations in seeing how each line item feature interacted with one or many of the other features in that list. Then when testing, there'd be another layer of questions and impressions I would have, informing new tests and increasing my perceptions of what it did.

So when it came time to suggest a theme as host for Weekend Tester session #32, I knew it would be interesting to see what they thought of the complex Mifos application.

I gave them a charter:

"Read the new user manual – http://en.flossmanuals.net/bin/view/Mifos/WebHome. Also read the welcome (http://en.flossmanuals.net/bin/view/Mifos/Welcome) to learn a bit about the purpose of Mifos and pick one of the following sections to discover new bugs, usability improvements, and user manual improvements."

Testing wasn't all that important to me for this session, actually. I just wanted to discuss the notion of complexity and how it affected test planning.

But I didn't expect the conversation to be enhanced by the fact that we used a very SIMPLE application to collaborate our test notes. We used typewithme, a free online application that allows real-time notes to be displayed by whoever decides to type them in the window. It's a very simple application – nothing to set up, just a link to email anyone who you want to share the session. That's it. No layers of features, no permissions, no formatting to worry about, just open a new session, share the link, and type.

From that experience with simplicity and complexity back-to-back, 3 lessons about complexity emerged:

1) **Complexity involves many factors working at once**.

2) **Complexity involves being overwhelmed**.

3) **Complexity is an emotional relationship with the object of study**.

1) Complexity involves many factors working at once.

One tester said: *"I stopped reading the manual because there was too much to read (too complex). I felt I had to try the product to get context then re-read the manual. It was only at that stage that I could read the manual in detail to find issues with the wording, etc. I found the manual was non-technical, and I needed some technical content to understand how to enter data in the fields - eg. the limits of the fields, or field types, or basic date rules."*

Remedy:

Remember that this Weekend Tester session involved a live chat via typewithme, so it was easy to collaborate. It was in that public discussion where one tester suggested, "Get it to a base state and build from there."

This reminded me of the Stone Soup fable: a traveler comes to a village in search of food. Hearing that they have none, he asks for a pot and a fire to boil some water. He takes a rock and tells them it will magically make food for all to eat. He puts the rock in and invites all to taste it. "Boiling water?!?" says one, "I have a carrot. At least it will have some flavor." "Hot carrot water?!?" says another "I have a sweet potato I can add." And one by one, villagers begin to make a soup adding one vegetable at a time. By doing so, it becomes more complex than hot water. In this fable, the point **was** to add more layers (flavors).

There is a procedure to aid this kind of modeling. The "General Functionality and Stability Test Procedure" (GFS) is a testing and modeling process created by James Bach when consulting for Microsoft in 1999 to help them come up with a way for third-party software to earn the "Certified for Windows" stamp in 6 hours, no matter how complex the application. (For details, see http://www.satisfice.com/tools/procedure.pdf)

It involves doing a feature analysis (as I did above for Mifos) and categorizing features into two categories: "Primary" or "Contributing". A primary feature is one in which it is so important, that if it was broken it would make the product unfit for its purpose. A contributing feature either supports a primary one or stands alone, but if it were to have a primary flaw, the product would still be able to solve its primary goal.

2) **Complexity involves being overwhelmed**. During the Weekend Testing session with Mifos, a tester said: "The manual was too complex, but not the wording. It was written in a very basic way. It was just too much too soon. My brain was not prepared to read all that information. I had to try to learn it quickly in increments. That meant doing about 3 passes over the manual as I was learning the data entry form."

Remedy:

One way to fight complexity is to break up testing into time-boxed sessions. Instead of written test procedures or cases, you can spend time exploring a charter – a mission statement for a session. Each charter comes from a few minutes modeling the product (as I did above) and distilling test ideas into two or three sentences to guide the tester for the next hour or so. This limits the tester's attention on purpose. Working in session toward a mission is meant to focus and direct the tester – to simplify the complexity of the product so it's easier to test.

This is exactly why I wanted to start with a charter for the Weekend Testing session. I knew the testing was time-boxed and adding a time pressure to missions can make it seem complex. It can be overwhelming to meet a goal knowing that every minute must be used wisely.

Another tester said, "I find that learning in waves works best for me. I learn the structure of a problem/content first, then I learn the major components, and finally I fill in the gaps. I require my learning materials to be presented in a similar way."

The most common way to distill complexity happens every day on software projects -- get more eyes on the problem than just yours. Each person on the team takes a piece (a feature, for example) and makes it their own. Pairing, so one person isn't doing all the thinking and analyzing, is a possible technique to fight complexity. Have someone brief you, or make a bold boast and agree to train someone else on a feature you're confused about – that service mindset may inspire you to learn in a new way – to see it through new eyes.

3) **Complexity is an emotional relationship**: Whether it be confidence, safety, or familiarity, a tester told me, “You have to acknowledge the level of detail you are at, and move between your 'safe zone' and the complex zone. There is always a level you are confident with. If you freak out, then go back to your 'safe place' and start again.

Remedies:
That same tester said: “If we model at each level or state, then we improve our level of understanding, and the depth of confidence with complexity. It’s just like a baseball game, and you move from base to base, eventually scoring a run. Modelling helps us find a safe place when looking at complex pieces.”

I heard “safety” and “confidence” in that remark – two emotional considerations that might be necessary to help someone cope with data overload. Most of the testers in this session were not familiar with financial applications, but two of them were. Whether they were or not, whenever someone encountered an emotion of complexity, they seemed to let the group know, as if to set expectations correctly.

Diagrams or visual models are meant to intentionally limit the amount of information coming through. It’s a change in perspective. It’s as if you take a pen and paper and draw a simplification of the dials and gauges in an airline cockpit, attempting to describe its essential functions to someone who had never seen one. But upon seeing the model, they may immediately abstract it to something that’s similar to what they *do* know. They may even have an epiphany of some kind of analogy, like “Testing is like a bird looking for worms.” That kind of attempt to find meaning and familiarity turns complexity into simplicity.

With typewithme, people seemed not only delighted by its simplicity. They had mastered it in seconds because it was a familiar paradigm and did not restrict or force them to think in a way to which they were unaccustomed. In fact, many wanted to know **more** of its other features to the degree that when asked to do another Weekend Tester session, I made exploring its features the primary mission.

Conclusion

The experience with group testing Mifos showed me that though complexity is a perception, it is also a force of many factors acting at once, overwhelming you, causing an uncomfortable emotion which may promote a desire for safety and familiarity.

It’s easy to say “just take it one step at a time” in fighting complexity, but even to say that much means being aware of what complexity is for different people. Are they an airplane pilot, unafraid of all the dials and switches in front of them because they understand the value and purpose of each? Or could it be that under pressure with inadequate emergency training, that same pilot would become paralyzed because of emotional overload?

The experience of this overseas collaboration in real-time with a two real products demonstrated validated my first experiences with Mifos as a complex application, but it also gave me some insight by way of the three lessons above as a starting point to understand when and how to confront (simplify) complexity.

Coding vs. Scripting: What is a Test Case?

Sam Bedekar
sam.bedekar@microsoft.com

Julio Lins
julio.lins@microsoft.com

Microsoft Corporation

Abstract

Automation has its place in the test lifecycle, but when it comes to automation, the definition of a test case becomes a gray area. Is a test case an XML blob or other metadata that is interpreted by a harness; or is it scripted code (such as JavaScript or VBScript); or is it a function or set of functions within compiled code written in a language like C# or Java?

A test case can be any of the above. One could define a test case as a set of instructions to execute a task and verify its results. These instructions can be described in a variety of languages.
This paper presents our experience in defining and implementing a script language, its applications in our test scenario; as well as our experience in adopting the Microsoft .Net platform with the C# language to create a long lasting and reusable test model.

We started with a set of principles and goals in conjunction with a number of technical challenges to solve. We initially created a test scripting language known as the Simple Language for Scenarios (SLS). We used this language for 3-4 years. As the C# language matured, we were able to find ways to implement the benefits of SLS as well as leverage the power of a fully functional development language. We concluded that overall each approach has its unique pros and cons and it makes sense to use the approach most suitable for a given situation. However, overall we found that moving towards C# was the solution that would work in most situations as it provided us with majority of the functionality and added the structure of an Object Oriented Programming Language.

Biography

Sam Bedekar is a Test Manager at Microsoft, working on Unified Communications (UC) technologies. Over the last ten years, Sam has helped grow the Unified Communications space as it transitions from hardware models to software and services models. Sam has been passionate about pushing test technologies forward at the same pace as the UC sector moves. Sam has a B.S. in Computer Science from the University of Maryland, College Park.

Julio Lins is a Test Engineer 2 at Microsoft, working on Unified Communications technologies. With a background in development of large scale corporate systems and a B.S. in Computer from UFPE in Brazil, Julio has joined Microsoft 4 years ago into the Unified Communications group. Julio has been focusing on improving the test process through automation, new testing tools and techniques.

1. Background of this Case Study

We have been testing Microsoft Office Communicator 2007 R2 and beyond. Communicator 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, 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. The framework would have to handle testing this.
6. A transaction (or scenario) may involve multiple clients to complete.
7. The Communicator client uses a STA (Single Threaded Apartment) model of threading. We were constrained to run within this framework while controlling multiple clients simultaneously.

Sample Test Case

One typical scenario in Communicator 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.

Three elements 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 of the previous two. In both cases, the three processes communicate remotely.

1.1 Test Classifications and Individual Goals

When analyzing a test scenario such as the one above, one will usually have many options in terms of frameworks, languages and tools. Among those choices resides the decision of whether to use a compiled or a script language.

In reality, there is not a one-size-fits-all solution. In particular cases, scripts may be more appropriate. In other situations, writing compiled code is the only viable solution. To start dissecting the problem, we start by examining a few of the basic classifications of test cases and enumerating their goals.

1.1.1 Functional Testing

When we implemented functional test cases, we established a set of requirements for the code to be written.

1.1.1.1 Requirements

1. Re-use of Test Cases & Code
    a. Must be able to re-use test results for measurement of performance
    b. Test cases should be written such that they can run at various layers – API layer vs. UI layer vs. protocol layer.
    c. Re-use of test code (libraries) between testers & developers

    d. Tests must be able to run on various platforms -- Windows CE, Browser, Native platforms for re-use.
    e. Tests must be able to run on a Server topology as well as Service topology without changing test case code
2. Rapid Test Case Development
    a. Easy orchestration of multiple clients without each case requiring to implement remote communication on its own
    b. A standard language to be used across the team
    c. Auto-generation of test cases when feasible (from a model)
3. Reduce redundancy in test cases by having a test model based on product architecture
    a. Decide what gets tested at the UI level versus the API level (object model)
4. Continuously execute automated tests
5. Must provide a unified Test Harness UI for the developers and testers to run the tests
6. When the product UI changes, all of the test cases should not need to be updated; these updates should be insulated from each case.
7. Strive for end to end automation by building test support infrastructure
    a. Soft IP Phone (SIP) simulator on Windows Device Simulation Framework
    b. SIP Proxy to modify SIP packets for fault injection
    c. Image comparison tool (compare slides, Desktop Sharing input/output)

1.1.1.2 Resulting Principles

As a result of the requirements, the terms below define our test strategy for test case automation.

1. Ease of Use –
    a. Functional Tests will have one front end/launcher regardless of whether they are based on the product GUI or API
2. Rapid Test Case Development & Reusability
    a. GUI automation must generate GUI classes (e.g. ConversationWindow) from input files
    b. Multilayer architecture for the remote endpoints -- Abstraction Layer will provide higher level methods for transactional operations (MakeCall, AnswerCall, etc.) within a client, whereas the Test Library layer will provide higher level methods for transactional operations cross clients (e.g. EstablishCall(client1, client2)).
    c. Create a Communicator UI Abstraction Layer to provide abstraction to avoid changes in test cases when the product UI changes
    d. Employ Model Based Testing tools – Generate test cases for test data matrices and test state machines
    e. All test cases & infrastructure will be C# based using common harness functionality through trapping of function calls (*more details provided below*).
3. We will invest significantly in shared libraries for common automation goals
4. Use Lab Bench Harness *(Microsoft Internal)* to allow running specific suites and integrate with a centralized bench suite for scheduling/launching.

1.1.2 Unit Testing

On the unit test side, with focus on developers, our requirements are listed below.

1.1.2.1 Requirements

1. Easy to write, short, focused cases
2. Able to see the details of specific error codes, inspection of variables, etc.
3. Emphasis on being able to debug during execution
4. Must allow for re-use of unit tests into the basis of larger tests (Build Verification Tests, Functional Tests)
5. Must be easy for both developers and testers to add unit tests as appropriate
6. Run on as few machines as possible

1.1.2.2 Resulting Principles

1. Ease of Use
    a. Share same test harness as functional testing
    b. Run multiple clients on one machine when no dependency exists (for example, no video camera sharing necessary for the test case, etc.)
2. Debugging the code
    a. Allow for Visual studio or other debug tool for debugging during execution
    b. Provide “Attach Points” where the case pauses after setup
    c. Allow for hooks to dump variables and object properties to a log file from test code in debug mode
3. Use the same harness as functional test cases to launch unit tests

1.1.3 Stress & Performance Testing

The stress and performance testing requirements came primarily from the test team.

1.1.3.1 Goals & Requirements

1. Scalable Test Harness (low overhead in the orchestration engine)
2. Test Harness should scale to high number of clients per machine
3. Client Stress should target Race conditions on the client
4. Re-use results from functional runs for measuring performance

1.1.3.2 Resulting Principles

1. Focus on core areas
    a. Conversations Stress
    b. Publisher/Subscriber Stress
2. Stress tests won’t be “Functional in a loop” – they will take the Abstraction Layer and run it as a state machine

2. Test Environment

From the background and test classification, this particular problem space results in the following technical challenges:

- **Asynchronous API**: Because the underlying product API was asynchronous, test cases would be required to verify that expected events are fired with specific properties. One of the main design targets would be to make it easy to write test code as well making the code itself very readable.
- **Multiple endpoints**: Each test case may spawn multiple endpoints on the same or multiple machines.
- **Multiple platforms**: the test code would run on multiple platforms, including Windows desktop, Windows CE and Windows for mobile devices.
- **Multiple Test Modes**: The core of the test business code would be used in different configurations. In each configuration, the product business logic would behave slightly differently.
- **Different scenarios with code steps**: The framework should support a very different set of test cases reusing the same business model.
- **Logging**: Detailed information of many complex event data sets.

Some of these requirements are characteristic of large scale systems. In fact, this is exactly the problem we had at hand: how to design a test framework for a large code base containing core business logic.

3. Test Infrastructure Design

3.1 Infrastructure mechanisms

The diversity of scenarios needing to be covered resulted in many infrastructure components to allow running the code in multiple platforms and under multiple different constraints. The elements below are part of the test infrastructure developed for both the script and C# automation.

Endpoint hosting
Since most of the test scenarios use multiple endpoints, the test controller needs to be able to drive multiple processes. This requirement resulted in the creation two elements in the test infrastructure:

- RemotingService: a small startup process that runs on each test machine and listens on a specific port for requests. Once the test controller needs another endpoint, it will send a message to RemotingService process requesting a new endpoint on that machine.
- TestEndpointHost: a console application responsible for starting the logging components, loading the root object of the test endpoint and providing a message loop to run the test endpoint.

The combination of these two console applications offer a simple but useful mechanism to start and stop endpoints on test machines as needed by each test case.

Remote communication
The communication between the test case and its endpoints is done inter-process and usually also across multiple machines. To solve the issue of allowing inter-process communication we used the standard .Net Remoting API for the Desktop platform and an in-house implementation for the Mobile platform.

Event storage and comparison
One of the intrinsic characteristics of the Communicator implementation is its asynchronous behavior. All the application state changes are reported as events. Whenever the test code orchestrates an action, it verifies the execution of such action by checking what events have been fired after the action was taken. Our strategy for comparing events is building infrastructure to subscribe to all events, store information from the event arguments and the application state and then save the event representation in a list. Whenever the test case needs to verify an event, it will block itself (thread-wise) by specifying an event name and property values to the event list. The event list will either find the expected event or time out.

Thread switching
In some scenarios, especially on the Mobile and Silverlight platforms, the product code cannot be called into from another thread except by the thread running the message loop. In those cases, it is necessary for the test code to switch threads before making an action. The test infrastructure solves this by posting messages into the message loop which allows a particular function to be executed in the main thread of the endpoint.

3.2 Pre & Post Execution

From the requirements, one clear theme that emerges is the need to run cases in a vast number of configurations. The ideal goal is to have the same test case code run in several configurations without modification.

To facilitate this, the test harness should be able to run a set of steps in the beginning of a test case or even before a single line of code within a test case. Finally, a set of steps also needs to be performed after a test case is complete. This allows us to insert logic that gets applied universally without each test case author needing to write code or call common library methods.
Reference: An example of this includes Application Verifier (Microsoft Technet).

3.2.1 Pre-Execution

Several common steps can be done before a test case actually starts. In some cases, some can be done before a certain step of the case. Our test harnesses performed these tasks in a common place, alleviating the need for each test case to implement these steps. These included:

- Logging environmental information: operating system, Office version, etc.
- Execution Information Logging: The harness takes care of housekeeping tasks such as log collection, results uploading, etc. without requiring users to call even a single function call. Each line in the case is logged along with a line number without the test case writer writing a single "Log" statement.
- Thread Synchronization: Ensure that test steps were executed on the correct thread without blocking any threads when waiting for events
- Custom Remote Communication: the engine abstracts out remote communication technologies so that we can run distributed system tests and have the test case be agnostic of the configuration in which it is running.
- Global changes in the harness can be used to modify the behavior of each case without modifying the case code itself. This helps us write and maintain hundreds of cases in a short period of time.
- Evaluation of API testing mode: If this same test case is being repurposed for API parameter testing, run the appropriate steps (as annotated in the case) in a loop and pass in various API parameters.
- Evaluation if a test case should be run based on environmental conditions: Certain cases may only be relevant in certain configurations (e.g. a voice case should only be run if the test account is enabled for VoIP communications).
- Real-time debugging/change of flow without the need for a debug environment (e.g. running in a lab environment)

3.2.2 Post-Execution

- Tallying/Logging test results
- Specific routines to run on Failure (e.g. "Clean-up" routines to avoid stale endpoints)
- Auto-analysis of log files for failing cases
- Copying/archiving of logs for facilitation of investigations

3.3 Creating an Intercepting Interpreter

A general solution for these requirements is to create an interception layer that can intercept each line of test case "code" and interpret it before execution. Early on, we found it to be an optimal solution to create a simple scripting language that would interpret the test case code and run the pre and post processing as part of the execution.

In general, the Test Library is agnostic of the infrastructure in which it is run. The Test Library is C# code that will perform operations, check events, etc... More details of this are covered in the section entitled "Interactions with C# Test Library." The C# code is then run by an intercepting interpreter that would perform the pre and post execution steps. In the case of SLS, the script interpreter that runs each SLS line would run the pre and post execution steps. In the case of C#, we used .NET dynamic proxies as our mechanism to intercept. Typically, dynamic proxies are used for custom marshaling of objects in a .NET remoting situation where custom marshaling is required (e.g. traversing special firewalls with a custom channel). We found that this similar mechanism would allow us to trap each method call on the test class and perform the pre and post execution steps in the C# engine beforehand. As shown in Figure 1, the test harness will instantiate the C# class in a dynamic proxy, which then gives it a chance to intercept each method call on that class.

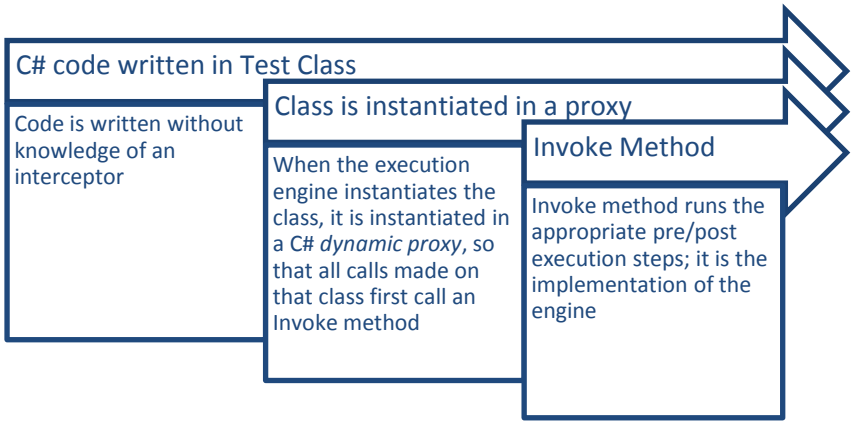


Figure 1: How an Intercepting Proxy is implemented in C#

4. Implementation of a Scripting Language - SLS

In 2003 we created a scripting language called the Simple Language for Scenarios (SLS) to fulfill these requirements. In SLS, we require users to write test cases in a particular text-based format (not XML based). This reduces coding-style flexibility, but provides us a single paradigm for writing test cases and a point of injection via the interpreter.

We use this intercepting point to achieve several goals – we were able to automatically marshal calls between different thread models, adapt existing cases for API parameter testing, and enabled security fuzzing without requiring each tester to have to write code to do so. For string injection, we wrote an API string testing class where the script engine could automatically call each method in a loop and pass in all possible parameters within the context of an existing case simply by annotating the “start” & “end” points in the test case where the engine needed to loop.

Additionally, with API testing, we were able to run a case to a particular point, and then bring up a UI to allow ad-hoc testing. The steps executed by the tester in the UI would then be written back as script language that could then be incorporated into the test case library.

We continue to use SLS in this form for a significant set of our automated tests today.

4.1 Brief Description of SLS

4.1.1 Constructs that had to be implemented

As part of authoring a new scripting language, there were several constructs from “full” programming languages that we brought forward into the scripting language:

1. Variables - the basic support of variables was necessary to avoid hardcoding strings
2. Arrays – array support was added to facilitate repetitive functions, like API testing
3. ‘for’ loops – For loops worked hand in hand with arrays
4. Function – Function support also helped consolidate code in a central place

4.1.2 Anatomy of an SLS Line

Goals

- For rapid test case authoring, spacing and brevity are important in an SLS line
- An SLS line is optimized to do multiple operations from a single line of code
- SLS was optimized to manage multiple clients from a single controller

The SLS grammar can be one of the following:

```
SLSLine = ExecutionCommand|VariableDefinition|Internal Command|Comment
ExecutionCommand=S_OK|IGNORE|E_FAIL|objectVariable ClientID [^] MethodName
Parameters
VariableDefinition=$VariableName=VariableValue
InternalCommand=@InternalCommand CommandParam1, CommandParam2, …
Comment=# Any comment text
```

4.2 Sample SLS File

Suppose that we want to test Group Instant Message –instant messaging across several clients. This requires the interaction of several clients, and verification of various events and exceptions on each client. With the underlying C# library (called the "Remote") and SLS, the scenario becomes easier to read and to script.

```
#
# MIM.sls - Test a Simple Multiparty IM scenario
#

# Create the Remotes

S_OK Client1 Create RTCV2Remote
S_OK Client2 Create RTCV2Remote
S_OK Client3 Create RTCV2Remote

S_OK Client1 Logon $server$ $user1$ $password1$
S_OK Client2 Logon $server$ $user2$ $password2$
S_OK Client3 Logon $server$ $user3$ $password3$

# Create a basic 2 party IM session

S_OK Client1 Invite $user2$ Session1
S_OK Client2 ^CheckSessionEvent SESSION_INCOMING
S_OK Client2 AnswerCall $user1$ Session1
S_OK Client1 ^CheckSessionEvent SESSION_CONNECTED

# Have the initiator invite a third party

S_OK Client1 Invite $user3$ Session1
S_OK Client3 ^CheckSessionEvent SESSION_INCOMING
S_OK Client3 AnswerCall $user1$ Session1

# Now both client1 and 2 should get a connected event

S_OK Client1 ^CheckParticipantEvent PARTICIPANT_ADD
S_OK Client1 ^CheckParticipantEvent PARTICIPANT_ADD

# Make sure Client2  can't add client3 since he's already in

E_FAIL Client2 Invite $user3$ Session1
S_OK Client2 CheckException ParticipantExistsException $user3$
```

Figure 2: Sample SLS file

One advantage of using an interpreter is the ability to do real time code modification, debugging, or stepthroughs without the need for a debugger on each machine. Interactive mode allows a user to step through an SLS file line by line, set execution breakpoints within an SLS file, skip lines, and more. It is similar to using a traditional debugger such as windbg on an SLS execution job.

When interactive mode starts, the line is displayed and a prompt is given to the user to choose the next action.

SLS commands mirror those of tools such as ntsd which are found in the Debugging Tools for Windows.

Table 1: SLS Debugger Commands

bp – Set a breakpoint	sxe – Enable Interactive Mode BreakOnFailure	i – Ignore/Skip the current SLS line	q – Quit the execution
bl – List all breakpoints	sxi – Disable Interactive Mode BreakOnFailure	k – Display current execution state (shows the line #, etc.)	? – display this help
bc – Clear breakpoints	p – Program Step (Execute one line)	g – Continue execution out of interactive mode	

4.3 Usage of SLS

4.3.1 Parameter Testing Mode

For our SDK release, we needed to test the inputs to the API. One large hurdle was that a particular API would only be callable when certain preconditions were met. For example, calling AddParticipant() on a conference session was only valid when the session was already connected. This context already existed in existing test cases. The Conferencing functional case knew of this order. We wanted to be able to reuse those functional cases and yet do API parameter testing. To facilitate, SLS provides an API parameter testing mode that would loop and pass in inputs to API calls based on *source* classes.
A SLS source class defines the inputs for a particular type. For example, the String source class returns various strings to be passed to APIs taking a string input. Examples include various size strings, null string, empty string, localized strings, URLs, html, xml, etc. as well as randomly generated strings for security fuzzing.

An API in a library is annotated with a C# attribute defined by SLS to declare the source class. In the example below, the Uri parameter of the Invite method is annotated to associate the string source class with the parameter. Because "uri" is the first parameter, the "$user3" variable in the invite method would be sequentially substituted with methods from the String source class. The sessionName parameter is not parameter tested. Finally, the test case would be annotated with the start & stop markers where SLS should loop.

```
[SLSUnitCategory("uri", "StringSource")]
Invite(string uri, string sessionName);
```

Figure 3a. Annotation on the Invite method in the Test Library method

```
# Create a basic 2 party IM session

S_OK Client1 Invite $user2$ Session1
S_OK Client2 ^CheckSessionEvent
SESSION_INCOMING
S_OK Client2 AnswerCall $user1$ Session1
S_OK Client1 ^CheckSessionEvent
SESSION_CONNECTED

# Have the initiator invite a third party

@BeginAPITestMode
S_OK Client1 Invite $user3$ Session1
S_OK Client3 ^CheckSessionEvent
SESSION_INCOMING
S_OK Client3 AnswerCall $user1$ Session1
@EndAPITestMode
```

Figure 3b. SLS script code using annotated Test Library method (Invite)

In this example, the Invite, CheckSessionEvent, and AnswerCall would be called repeatedly until all string sources were exhausted. New test cases wouldn't need to be written – the same functional test cases could be re-used. This helped us add substantial coverage with minimal overhead/effort.

5. Interactions with C# Test Library

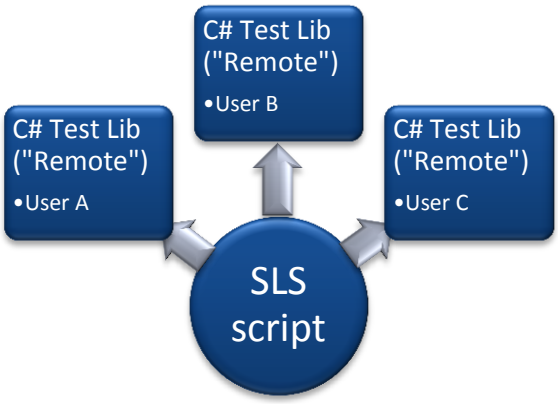


Figure 4: Overview of Interactions of SLS and the Test Library

5.1 Overview of the Test Lib

The Test lib represents the code of the main business logic of the test cases. It is implemented directly on top of the product code and verifies its expected behavior. Since the product is asynchronous in its nature, the purpose of the Test Lib is to make method calls into the product code and then verify that the expected events were correctly fired. So, the Test Lib implements a synchronous view of the product API. For example, the Test Lib is what implements the “Invite” call in the above group IM example.

The chosen test design is such that the Test Lib subscribes to all events of each object from the product API it instantiates. Whenever an event is fired, all of its properties and some of the application behavior are saved in the form of an event object. An infra-structure component named EventManager saves a list of events. In the above Group IM example, the “SessionStateChangedEvent” would be queued in the EventManager until it was checked by the test case code.

As another example on one endpoint, the Test Lib code that calls SignIn() in the product API, also calls the EventManager Wait() method, specifying a new event instance filled with the expected properties of the actual stored event. This is how the Test Lib code verifies not only that the event has been indeed fired but also its properties and the application state at the time of the firing. As a result, the Test Lib provides a synchronous SignIn() method which will block the caller until the correct event is fired, or a timeout happens.

The Test Lib is a highly reusable component. The great majority of the test cases use at least one or two methods in the Test Lib to achieve their particular scenario. The implementation of the Test Lib had to vary slightly to work best with SLS or C#. Below, we will go over the differences in the Test Libs.

5.2 Design Patterns of the SLS Test Lib

In the SLS library, the Test Lib is known as the “Remote” as it encapsulates one remote endpoint.

Container Class
The Remote is primarily a container class. It contains references to instances of API objects (such as CommunicatorClient, Conversation, etc.). One instance of the Remote is used to drive one instance of an endpoint. In addition to the references, it contains lists that hold incoming events and exceptions. Events

and exceptions can then be verified with functions that traverse the lists. Additional informational data is held, such as usage counters (used during stress) and data about attempts of certain actions made on objects (which can be compared to the actual state of the object to detect errors).

Method Wrapper and Helper
In order to keep scripts brief & concise, the script language cannot have much "logic" in the script layer. Thus the primary way of using the Remote is to call wrapper functions that add additional logic. Each wrapper contains a try/catch block that will queue any exceptions. The wrappers are identical to the underlying API except for the addition of return values. For example, suppose you have a buddy object that contains a name, data, and a URI. An API called "GetBuddy(string uri)" returns a buddy object. The Remote's corresponding wrapper function would be -- GetBuddy(string uri, string expectedName, string expectedData). The wrapper function would call GetBuddy(uri) on the Collaboration object, and query the returned buddy object for its name and data properties, and compare it with the expected properties. If all values match, then the wrapper function would return success.

Uniform Return Values
All methods on the Remote will return a type known as RemSummary (Remote Summary). The summary describes the result of the method call, as well as reasons why it failed (if it failed). With all methods returning the same type, SLS or any language can use this information to display uniform information and keep uniform test result counters.

6. Transition Period: Communicator SDK Test Framework

Although we were successfully using SLS for testing several versions of Communicator, an opportunity arose to evolve and tweak our strategy. We had a fundamental design change in the underlying product API that existing test cases were written on top of. The new Communicator object model code was being designed to be offered as a product with the main objective of exposing the Communicator business logic in a simple to use fashion. In contrast, the previous API was designed as a low level session API. Neither the existing test cases nor the existing test code could be reused across product sets without extensive rewrites.

Moving the test code to the new API was required since the new design would be shipped as a product. This would have the advantage that all test cases written on top of it would test not only the Communicator business logic but the API itself. In addition to that, once the product code shipped, testers would benefit from the API being stable. Any changes made to the API would be made to maintain compatibility with older API versions.

The fact that the test cases would be written on top of a common and stable API brought the reuse of test code into much evidence comparing with the test code requirements for Communicator 2007. This led the test team to start analyzing pros and cons of a scripting language versus a compiled language.

At this stage in the product, the size of the test code & test team also grew significantly. Approximately 30 test developers were allocated to the task. This imposed a lot of coordination and need for configuration management.

The strategy adopted to implement the new test cases included:

- **Using the C# language**: since reusing the test code was one of the main targets, using a fourth generation compiled language brings many benefits compared to script languages, in terms of development of components, build verification and compatibility with external tools and frameworks.
- **Using an in-house test harness**: Due to existence of an in-house well-developed and complete test harness, we chose to use it to run our test cases.
- **In-house development of test infra-structure**: the nature of the required test environments required several components to support the test cases.

7. Design Patterns of the C# Test Lib

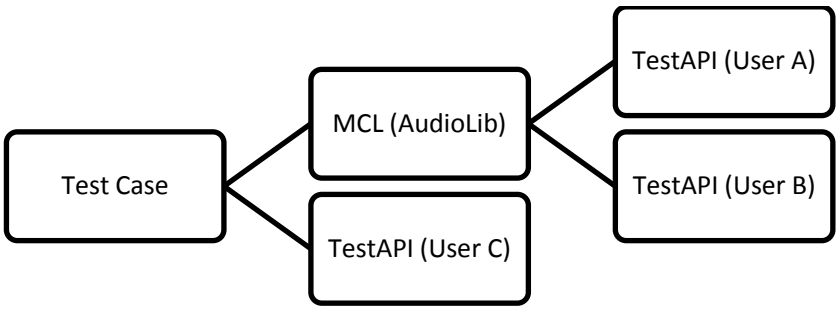


Figure 5: Overview of the C# Test Lib Design Pattern

The Test Lib code implemented was divided into several components:

Multi-client library
The customer scenarios for Communicator often involve a party of two or more users, represented as endpoints in the test code. Each endpoint is a running instance of the product business logic and usually runs on its own machine.

A great number of test cases involve taking actions across multiple endpoints. Imagine an audio call scenario, in which user A calls user B. The test steps involve making a call from endpoint A and accepting the call from endpoint B, and then verifying the call is actually connected on both sides.
These four steps can be represented as an EstablishCall step, a set-up present in any test case that requires an existing audio call to run a particular scenario.

The multi-client library contains a set of reusable scenarios across multiple endpoints. Most of the code reusable in this layer is associated with a feature or with a class of test cases. Nevertheless, test cases sharing multi-endpoint steps are fairly common given the nature of cross feature integration of the product.

Test Lib
The Test Lib in the C# solution is called the “TestAPI.” Like the Remote, it wraps underlying API objects, handles events and exceptions, and aggregates multiple actions into one where appropriate. One advantage of using C# is that we utilize object oriented programming. The Test API and Multi-Client Library both use inheritance to avoid duplicate code.

Test cases
A test case is defined in our framework as a method in a class belonging to test module. Each test class represents a suite of test cases and the set of classes represent a module. Both the module and suites have a set-up and tear-down mechanisms.

The test method returns a pass/fail value. The implementation of the method uses the multi-client library to achieve the pre-requisites of the test case and then usually the specific test verification is done directly through the test library.

Reuse is still present in the test cases to some extent. Common functions among a class of test cases are shared in library specific to those test cases, which can be implemented as a separate component or simply helper methods.

In terms of software architecture, the structure of the test library and the multi-client library are well controlled and code reviewed, while the structure of the test cases library is open to the tester implementation.

7.1 Revisiting Pre and Post Processing

In the C# Test framework, we have made additional enhancements to the pre & post processing.

Module and suite level
On the module and suite level, pre and post processing is used as a mechanism for setting up and finalizing the test cases. This is implemented by features offered by the in-house test harness through annotations in the test code (method attributes in C#).

7.1.1 Method/Type level

To achieve a similar level of common infra-structure running behind the scenes as was present in the SLS script language, the test components make extensive use of the RealProxy feature of the .Net platform.

The RealProxy class allows the runtime generation of a Proxy object, which intercepts method calls to the associated business or test object. The dynamic proxies are used on both the test case side as well as on the endpoint side. One may interpret side as the hosting process of the code.

Automatic Logging
One of the first uses of the dynamic proxy feature was automatic logging of method calls. For each method call on the Test Lib, the logging proxy registers the type and method name, plus the value of each parameter.

This is extremely important for test failure investigation. Having each method log the parameter values manually would be extremely costly, whereas the proxy allows this to be done at infrastructure level. The information stored in the log file helps identify the root cause of the failure and the source of a bug whether it is in the test case or the product code.

At first the automatic logging was implemented on the endpoint side to save each call into the test library. Later it was also replicated on the test case side (hosted on the controller machine). Having automatic logging on the test case side was added due to the lack of support for the RealProxy feature in the .Net Compact Framework.

As a derived benefit from implementing automatic logging on the test case side, the method call in the log also contains the endpoint hosting that instance of the test library. This simple piece of information greatly facilitates the investigation of the cause of a failure.

Remote communication
One of the target platforms for our test cases is Windows for mobile devices. The device hosts the process running the product code together with the Test Lib. The test cases and the multi-client library run on the controller machine. The .Net Compact Framework does not support .Net Remoting, the main mechanism for inter-process or inter-machine communication in the .Net Platform on the Desktop.

The lack of support for object oriented remote communication left the test team with two options: re-implement the test code on the mobile platform or implement a similar remote communication mechanism on top of the .Net Compact Framework. We have chosen to implement an XML-based serialization component which implements similar functionality as that of the .Net Remoting API. On the test case side (Desktop) a dynamic proxy would intercept a method call and serialize it to the mobile device. A socket server on the device would parse the XML, reconstitute a method call and finally make the call into the actual object. The implementation enables the test code to transparently make calls into objects running

on either a mobile device or another PC desktop. As a result, the same test code was shared among the Desktop and Mobile platforms.

7.2 The experience of the process of implementing reusable test code

The main advantage of moving from a script language to a compiled language in our scenario is the facilitated ability to reuse code. Reusing product code is easily justifiable by the clear benefits and cost savings related to not implementing the same functionality more than once, or having to test it more than once. However, since the test code is not the product itself, our experiments pointed to a trade-off between reuse and cost of the development and stabilization for the test code.

A failure in a test run caused by product code is a payoff of the investment in testing it, while a failure caused by the test code itself is an added cost to the test effort. While it is still clear that reusing test code has benefits associated with it, it is our intention to share what it took for us to reduce the number of failures caused by changing common test code.

7.2.1 Configuration Management

This is one of the most important aspects of our test development. To establish an environment in which many people change the same test code base, not only version control is necessary, but also code reviews, a strong build system and a set of test cases to serve as smoke tests for the test cases itself.
It is important to reinforce that failures caused from the test code itself are simply added to the overall test cost of the project. The more test code is reused among different scenarios, the more this type of failures may happen, unless there is some process to keep changes controlled.

Among the practices we used are:

- **Code reviews**: test peers need to approve changes to the test code before any changes are committed to the repository. We established virtual teams which would be responsible for code-reviewing a set of components in the test code. This allows us to detect impacting changes and assess the consequences before the check-in. In practical terms, a test owner would be vigilant of the code his test is using to prevent an external change from affecting the result of his feature tests.
- **Strong build system**: Check-ins into the repository trigger full builds of all components. This way breaking changes to the code are rapidly detected by the build system. It is also part of the process for the tester doing the change to build locally and do buddy builds (move the changed code into another machine and build it from there).
- **Run smoke tests**: It is the responsibility of the tester making changes into common test code to verify if that change will impact other feature test cases. This can be achieved by defining a reasonably small set of tests which diversity enough to cover the most common features.
- **Pre-check-in test bench**: as an evolution to the points above, a system was developed to allow testers to submit the test code for an automatic build and run of smoke test cases before checking-in the code. This tool is also integrated with the code reviews. The process for checking-in changes requires, first, getting approval from the test peers; then using the tool to validate that the changes do not beak the build or break the smoke tests.

We believe that using a test framework built on a compiled language has helped us achieve such a complex process of reusing the test code that supports our long term test strategy, given the complexity of our test cases.

8. Comparison between scripts versus a compiled language

While one could easily have a full article comparing scripting languages to compiled languages, this section actually presents our own experience, with gains and losses, while using both strategies at different points in time.

To establish a process of comparison, the next sections analyze the test implementation using the SLS script language and C#, from different points of view.

8.1 Portability/Reusability of test cases

In SLS, the portability of test cases to different languages, new technologies, etc. was very low – as the test case library grew, it became very specialized within the SLS framework. As new people joined the team, they would have to get ramped up on SLS, the libraries, and the lab bench execution framework. This became an ongoing cost. In most cases, modifications to the SLS engine itself were not necessary. However, in the event that new support was required, only the original contributors to SLS had a working knowledge of the SLS code and had to be called in. The test case code was highly reusable to other cases within the SLS framework; but if someone writing a C# app wanted to use a SLS library function, it could not be called from C#.

Users of the C# framework faced a similar ramp up time initially on the test libraries and case structure. Because C# allows great flexibility, they would also have to familiarize themselves with the coding style and expectations of the team. However, the cases were highly portable; a new member taking over an area could immediately understand the case code and be able to trace through to the library layers. Executing and debugging was straightforward (though not necessarily easy). One hidden ramp up cost was that the pre & post conditions executed by the real proxy were not obvious to someone taking over the code; this centralized logic would have to be referenced in the RealProxy implementation. This "implicit" engine was a hidden component in the end-to-end lifecycle.

8.2 Productivity

Productivity was an area where SLS had a significant advantage over C#. The cost of writing a case was very low as the syntax was brief and the interpreter would run all the pre/post execution steps automatically. A script for a scenario could be created in a minutes. This allowed for a large volume of test cases to be created in a brief amount of time. In C#, the iterations required to create a case were higher due to build steps, debugging through a full debugger, etc. The proxy work to do automatic logging, etc. mitigated some of the costs.

8.3 Configuration Management

Source Control – We used the same source code management system for both solutions. Code reviews and buddy builds were mandatory prior to check-in. History information, branching solutions, and bug fixes followed the same workflow for the two solutions.
Builds – Building was non-existent for SLS scripts as there was no compiler. This was a disadvantage as any errors would only be caught at execution time (whether the error was added into the test case itself or if someone was changing an underlying library). This meant that the errors were caught further downstream, increasing the cost of bringing it to a resolution. The C# code would be built in the buddy build system, and any break would be caught prior to check-in.

8.4 Integration with other tools

As SLS code was interpretable only by the SLS interpreter, it could not be reused by other test harnesses. One provision we considered was to have the SLS interpreter create stubs for each SLS library method which would internally call the interpreter; however we did not get around to implementing this.

9. Conclusion

Each strategy served its own purposes and gave us different benefits. At the end of this exercise, we've realized that a test case can be script or code. As we faced different challenges, different solutions were

more appropriate. However, we did find several common themes emerge that helped us shape our solutions. Pre & post processing was crucial to reducing duplicate code and test case development time. Scripting allowed for rapid test case development but reduced reusability. Object Oriented Programming patterns helped speed up test library investments written in C#. After experiencing both, we've found that in the long term C# is our best choice due to its flexibility and the fact that most advantages of SLS could be implemented in C#.

We also learned that the test case development lifecycle investment is as important as the language chosen. Code review, quality gates, and configuration management is crucial to maintaining quality and long term maintainability of the code.

10. References

Debugging Tools for Windows. (n.d.). Retrieved from Windows WDK: http://www.microsoft.com/whdc/devtools/debugging/default.mspx

Microsoft Communicator Team. (n.d.). *Group IM* . Retrieved from http://office.microsoft.com/en-us/communicator-help/send-an-instant-message-to-a-group-HA010290007.aspx

Microsoft Technet. (n.d.). *Application Verifier on Microsoft Technet*. Retrieved from Microsoft TechNet: http://technet.microsoft.com/en-us/library/bb457063.aspx

11. Appendix A: SLS Keyword Syntax

Table 2: Variable Usage Syntax for SLS

8.4.1 String Variables	8.4.2 Object variables (o:)	8.4.3 Setting a variable as a Client to be operated upon
Command Line Parameter /D SLS File Definition using $ Configuration File Definition Environment Variable	Object variables can be used to store non-string objects that are returned by	@ASSIGNCLIENT @RELEASECLIENT

Table 3: SLS Keywords

8.4.4 Test Case Management and Metadata	8.4.5 Variable Manipulation & Generation	8.4.6 Network Information & Operations
@TESTOPENIF @TESTOPENEX @TESTCLOSE @FUNCTION, @ENDFUNCTION @SECTIONOPEN @SECTIONCLOSE @BUG @META	@TOSTRING @GENSTRING @CHANGEURICASE	@GETFQDN @GETHOSTNAME @GETLOCALIP @GETSECONDLOCALIP @GETIPFROMHOST @GETLOGGEDONUSER @GETLOGGEDONDOMAIN @OPENPORT @CLOSEPORT
8.4.7 Helper Methods @PRINTF @COPYFILE @DIALOG @SLEEP	**8.4.8 Loops, Conditions, & Flow Control** @LOOP, @ENDLOOP @CALL @CALLIF @CALLIFNOT	**8.4.9 SLS Execution Behavior Modifiers** @SET @ASSIGNCLIENT @RELEASECLIENT @GETREFERENCE
8.4.10 Stress Mode @CALLSTRESSVARIATION @STARTSTRESS, @ENDSTRESS	**8.4.11 Unit Test Mode** @LAUNCHUNITTEST **8.4.12 Math Operators** @ADD @MULTIPLY @SUBTRACT	**8.4.13 File/Environment Information & Manipulation** @GETTIMESTAMP @GETENVIRONMENTVAR @GETVERSION @GETASSEMBLYVERSION @GETPLATFORM @GETRAMINFO

Web Test Automation Framework with Open Source Tools powered by Google WebDriver

KapiL Bhalla, Nikhil Bhandari
Engineer, QuickBooks Online
Intuit, Inc.
Kapil_Bhalla@intuit.com, Nikhil_Bhandari@intuit.com

Abstract

Building test automation harness on open source libraries often causes more pain than pleasure. Factors like application complexity (variety in web elements used on a web page, nested frames, iframes, etc), project scope (number of web pages), test automation team size, learning curve of automation libraries, etc… lead to high maintenance and less adaptable solutions. In such cases principles of abstraction and design patterns come to rescue and help us attain better maintainable, adaptable and thus sustainable solutions. This paper will demonstrate how application these principles of abstraction and design patterns can aid us in building flexible test automation harnesses.

When we looked at QuickBooks Online for first time for automating its testing we knew that we were facing a daunting task. Some of the challenges we were confronted with were supporting automation for more than 100 web pages, carrying out testing on modal windows, handling AJAX controls, testing Java Script error handling, coping up with constantly changing product and User Interface with changes going into the web application once every 6weeks, managing contributions from geographically spread teams in US and India. Our exercise evolved with time into an operational test automation harness.

This paper will provide some context on various factors which make tailoring a test automation harness challenging and measures which will make your solution work. We will also propose a Blue Print for test automation harnesses.

Biography

KapiL is a Quality Engineer in QuickBooks Online and responsible for test automation. He has been with Intuit since Jan 2008. He received a Masters of Computer Application degree from National Institute of Technology, Karnataka India and a B.Sc. degree in Computer Science from University of Delhi, India.

Nikhil holds an Engineering degree along with 10 years of experience. He is currently working with Intuit as a Staff Software Engineer - QA and has previously worked with Oracle, McAfee & Satyam Computers in Bangalore, India. He has been speaker at STARWEST Conference 2008 (USA), Free-Test Conference 2009, 2010 (Norway), Step-In Forum Evening Talk (India).

1. Introduction

This paper shares learning from journey of building user interface test automation harness for over a decade old web product with constantly changing user interface and geographically separated teams testing the web application.

The requirements that we faced were making test automation harness simple for quality engineering team and demonstrating to the management that in long run changes to the web application can be accommodated in test automation harness at a low cost. Making the test automation harness maintainable, flexible and work across different types of user scenarios in the web application were the prime challenge. This required handling variety of web elements like radio buttons, select dropdowns, check boxes, text boxes, frames, nested frames, etc… and simulating user behavior using mouse actions and keyboard actions. While most of the interaction with the web elements was solved by Google's web driver we were left with the bigger requirements making it easy for test automation engineers to use the test automation harness and to make it maintainable. Let us see in detail how we solved for these requirements.

2. Background

The target application in our case was an accounting web application. Requirements included automating testing across – multiple browsers like - Internet Explorer, Firefox and Safari on windows and Mac operating systems. The product spanned over 100 web pages, and multiple web technologies like HTML, Java Script, Flex, CSS, etc… That have gained acceptance over the last decade. Additional challenges included enabling multiple members to contribute to automation testing without bumping into each other's domain specialization, and to be able to achieve code/ workflow/ test reuse wherever possible.

The dynamism in requirements came along when some of the backend services used were changed in initial months, presenting multiple web pages doing the same redundant things (example: login) in the application. Multiple login pages demanded test automation harness to provide support for both login pages at run time this was made possible by using factory pattern.

Choosing Google Web Driver as the backend library for the test automation harness required the automation team to spend time on learning and understanding the libraries. To minimize the learning curve we decided to hide web driver libraries and provide simple APIs for test automation engineers to use.

3. Understanding the requirements and Meeting them

Apart from the default requirements of executing automated tests across various browsers on multiple platforms we were confronting few other requirements. These requirements are:

1. Easy to Use test automation harness / Libraries
2. Easy to learn test automation harness / Libraries
3. Reusable Code/ Workflows/ Tests
4. Maintainable test Automation Code
5. Support for multiple contributors for the test automation code
6. Flexible, loose tool coupling

Let us dive deeper into these requirements and see how they were met.

3.1 Meeting Requirement: Easy to Use test automation harness

What could be one of the easiest usability cases that a test automation framework can provide today? You may be right with your answer *"Record and Play"*. It is true to a large extent that recording is simple way of generating code/ script and there is no doubt in that. There is usually some expected effort involved in refactoring these recorded scripts into some kind of 2-level / 3-level architecture to ensure maintainability. If refactoring is not done after record and play and your application is of reasonable size and is evolving on UI front then maintainability goes for a toss. On the other hand test automation harness / frameworks demand refactoring test automation code upfront- and the gains of abstraction are harnessed later in the life cycle of test automation code by means of reduced maintenance cost.

The target that we set for our self was exposing Java APIs for automation of possible actions a user can carry out on the web pages of our application. This is enabled by modeling the web pages in Page Object classes and the possible set of actions on the page as methods of the Page Object class (Standard model for UI automation). Let us look at this by an example,

```
// This is a sample Page Object representing a standard login page.
Class LoginPage{
        …
        private username = null;
        private password = null;
        private login = null;
        …
        public void setUsername (String username){
                username.sendKeys (username);
        }
        public void setPassword (String password){
                password.sendKeys (password);
        }
        Public void clickLogin (){
                login.click();
        }
        …
}
```

Some Benefits of this approach include limited impact zone for UI changes and clear separation between test code and web page model. On the downside modeling web application in page object initially consumes time.

Modeling web pages in page objects provides level -1 abstraction of the web product that is being tested. A clear benefit for test automation contributors is test case/ workflow will look similar to a <OBJECT>.<METHOD>(<PARAMETERS>) composition and assertions if any. You will shortly see how we build more on this abstraction. For example: a test workflow for successful login will look similar to,

```
public void testLogin(){
        LoginPage loginPage = new LoginPage();
        loginPage.setUsername("KapiL");
        loginPage.setPassword("Bhalla");
        loginPage.clickLogin();
        // code to assert that we landed on correct page post login
        …
}
```

With the specifics of the automation libraries encapsulated into methods of web page objects a test automation engineer gets to use simple java methods to simulate user actions on the web page. This makes the test automation harness relatively easy to use.

3.2 Meeting Requirement: Easy to Learn test automation harness

With the page objects in place, test automation engineer is largely saved from learning libraries of automation tool being used. This reduces the coding effort, learning effort and required skill expected from test automation engineer. By generating Java Docs for all web page objects in the test automation harness easy to read documentation can be provided to the test automation engineers, this will further aid a test automation engineer in learning the test automation harness.

3.3 Meeting Requirement: Reusable Code/ Workflows/ Tests

Fall back in your chair and imagine the login page of your favorite application (let us assume that to be *gmail)*. Given the task of automating the following 3 test cases:

1. Successful login
2. Wrong password attempted, and
3. Password left empty while logging in

What would do? Would you like to reuse the following 3 lines to code?

```
loginPage.setUsername("<your-user-name>");
loginPage.setPassword("<password-content>");
loginPage.clickLogin();
```

Or will you prefer a parameterized method called loginAs(username, password) which executes the above mentioned 3 lines of code? We will refer to methods like loginAs(…) as workflows and they help us grouping together common code that needs to be executed several times, this is workflow reuse.

Now consider that you want to login into gmail and test functionality of some cool features of gmail and while doing that you want to test login functionality every time. Given this requirement you may choose to put all assertions and validations along with the workflows that you grouped in methods or you may directly reuse your parameterized testSuccessfulLogin(…), this kind of reuse is referred to as test case reuse. Doing so not only helps us reduce the coding effort while automating but it also ensures that more frequently used components of your application are validated in proportion of their use.

Reconcile

Taking few moments to reconcile what we have till now shall be of use,

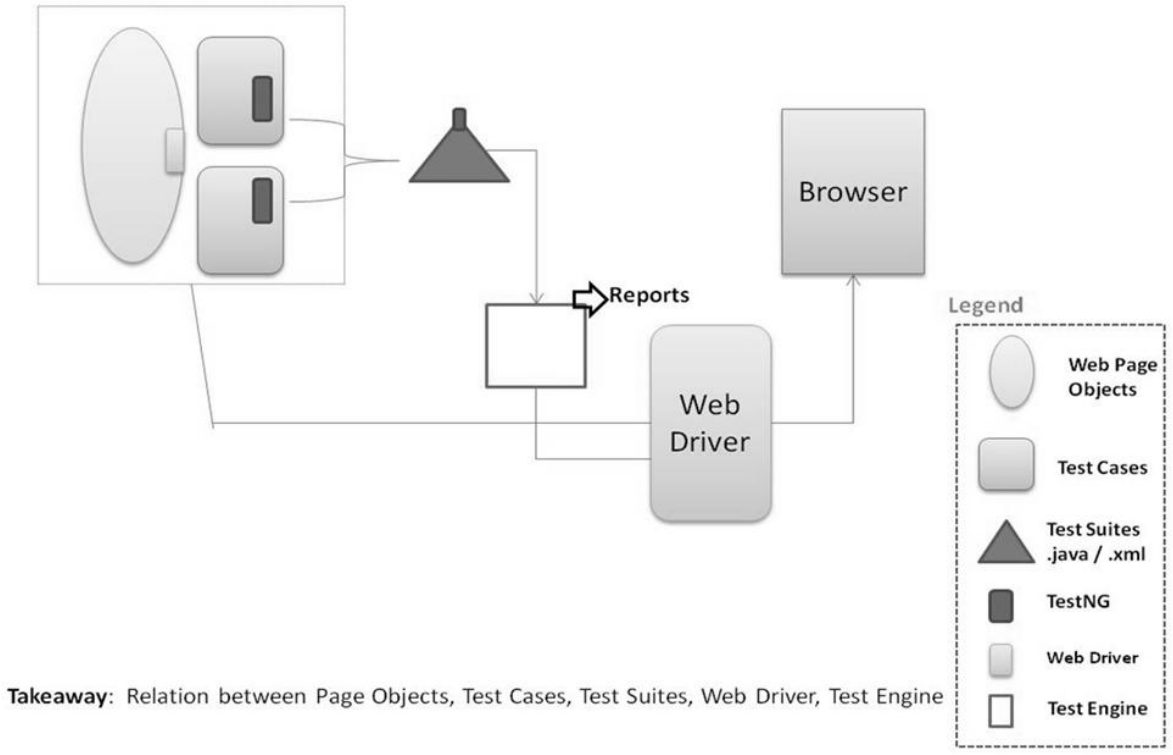


Diagram1: Represents architectural relation between page objects – workflows – test code

The above architecture diagram summarizes how web pages of your web application are modeled by page objects, how page objects are used by the workflows, and how test code use methods in page object and/ or workflows and/ or other test code.

3.4 Meeting Requirement: Maintainable test automation harness

If done well this shall be one of the major cost savings factors in long term. Historically speaking 'species adapting fastest to change has been most successful in surviving evolution', given that the only constant in evolving software is 'change' one of the biggest challenges for us was how open we were to changes being made in UI of our web application.

The change can be a simple change of attribute of an element on the web page (say an ID) to introduction of a parallel web page offering exactly same functionality as the one before (say a new version of Login page being tested for Beta experience). Let us list down some of the possible changes that are common to UI of a Web Application:

1. Changing attributes of an existing element on the web page
2. Existing elements being deleted from the web page
3. New elements being added on the Web Page
4. Introduction of a new page replacing the old page
5. Introduction of a new page along with the old page

The list of possible changes only grows during life span of a web application; this makes handling change not only important but critical. Diving deeper will reveal how we handled change. Architecture shown in

diagram 1 reduces lots of rework by enabling interaction with elements on a web page in object oriented manner, but there still exist few things which can be improved upon. On a closer inspection we found:

1. API calls within page object that bind us to the tool we are using (in our case WebDriver). Example:
    a. driver.findElement(By.id("<AN_ELEMENT_ID>"));
    b. element.sendKeys("<KEYS_TO_TYPE_INTO_THE_ELEMENT>");

2. Tight coupling of web element attributes to locate the web element on the web page. Example: IDs being used to locate the elements on the web page. "<AN_ELEMENT_ID>" in the above example.

This demands abstracting out the tool specific code from the page objects and abstracting out the attributes used to locate the web elements on the web page we will see how this was solved.

Abstracting out the tool/ library (WebDriver) from the page objects, demanded creation of a utility that will find elements on the given web page once provided with the attributes of the web element. All elements being interacted inside methods of page objects of the test automation harness are required to use this utility to locate the web element on the page.

Abstracting out the attributes of web elements from page objects confronts us with 2 choices:

1. Pass the attributes at runtime when test code is using the methods of the page object.
2. Load and make available elements on the web page when the page object is instantiated or upon use of an element.

This opens up another choice between Lazy loading and Eager loading of attribute specific data into the page object. With the classic tradeoffs of each we leave this choice to the inspired engineers from this paper. We choose option 2 and persisted attributes of the web elements in configurable 'Web Element Repository Files'.

Abstracting out the attributes of web elements from page objects offered tremendous flexibility while making the test code ready to accept changes in application UI, freedom from recompilation of page objects in case of change in attributes lead to reduced cost in handling one kind of change.

These two abstractions - test tool abstraction and web element attribute persistence - brings us to the revised architecture of the test automation harness,

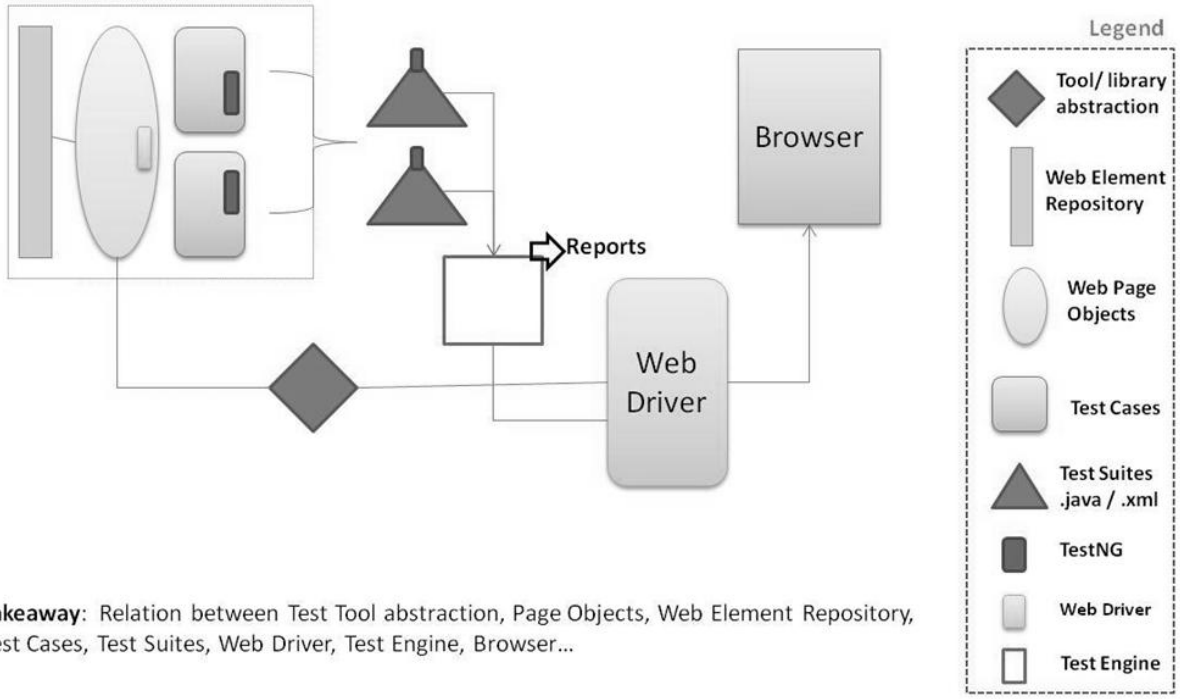


Diagram2 – Adding test tool abstraction and web element attribute persistence to test automation harness

3.4.1 Changing attributes of an existing element on the web page

With attributes of elements on the web page being persisted in configuration files/ database it is easy to make changes to them. The required changes to the attributes of elements like change in ID, finding the element using NAME, etc can be made by updating the respective element field / record in the persistence layer. Now changes to attributes of elements are not expensive and don't require compilation of page object or test automation code.

3.4.2 Existing elements being deleted from the web page

If an element is deleted from the web page then the corresponding entries in persistence layer needs to be deleted and methods using the element in the page object need to be updated. This kind of change can also impact automated test code written using methods of the corresponding web page object where the element was showing up. This is an expensive change and needs to be handled at all levels.

If the deleted element is replaced by a similar element then the web element mapping between the persistence layer and test automation harness can be reused and the impact can be less.

3.4.3 New elements being added on the Web Page

On addition of new elements on a web page we will need to add entries in the persistence layer and update the web page object to make available the set of possible actions on the web page using the new web element. This is also an expensive change and is likely to be made at all levels in the test automation harness.

3.4.4 Introduction of a new page replacing the old page

On addition of a new page replacing an old web page we need to provide support for the new page at page object level. If the page offers same functionality which was being provided by its predecessor then we may use the same page object, in this case change will be required at web element persistence layer and in body of methods of the page object.

3.4.5 Introduction of a new page along with the old page

When we need to support multiple pages offering the same functionality in the web application and we are not sure on which web page will appear on runtime in the workflow that is being automated, factory pattern comes in handy. In this case we have support for both web pages in the test automation harness at all levels, based on which page appears in our workflow we detect the page and get object of the respective page from the page factory in the test automation harness. This is relatively less likely to happen and factory pattern is one elegant way of handling the requirement of supporting multiple pages providing same functionality in parallel.

3.5 Meeting Requirement: Support for multiple contributors

Enabling multiple engineers to contribute in test automation is a well known requirement that we get when we set ourselves with the task of automating testing for a monster sized application. But with this requirement comes responsibility of managing all cumulative code that engineers write. Giving full freedom to modify or add code in any part of automation can be recipe for code maintenance nightmare, and too limited access can lead to parallel test automation for overlapping areas in the web application.
It is too early to comment on ROI of our implementation but sharing what we chose to do is precisely the reason this paper is being written.

We have chosen to arrange our test automation effort in alignment with the team structure of the development/ quality engineering team. The development/ quality engineering team is aligned according to areas/ features of the web application, with this team structure engineers working on particular areas of the application hone their skills to become domain experts. Packaging the test automation harness code in alignment with areas/ features of the web application empowered contributors/ domain experts with much needed independence while automating the tests and abstraction to others from test automation work being done on a specific feature of the web application. Some of the application areas and packages (java) in test automation project are:

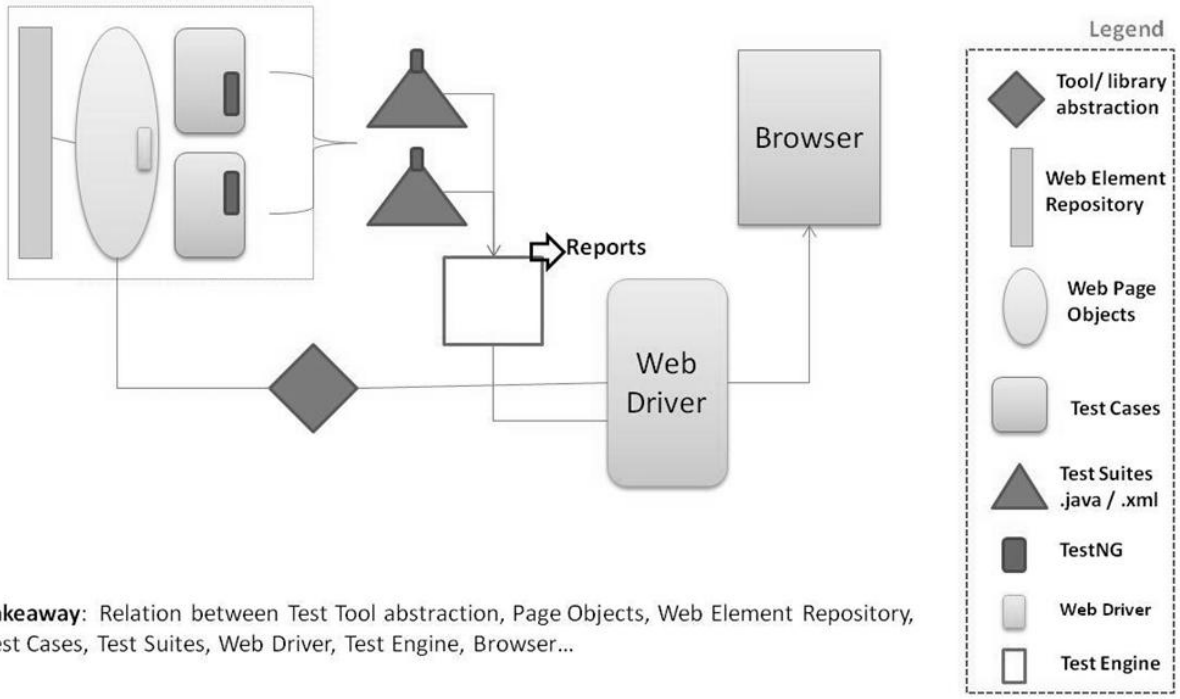
1. Login / Logout
2. Home Page
3. Administration Pages
4. Application Preferences, etc…

3.6 Meeting Requirement: Flexible, loose tool coupling

Separation of automation code into Page Objects and test automation code makes test automation code free from tool specific code. To further reduce the coupling of tool/ libraries we build a core class which carries out interactions with the elements on the web page, this is exposed by several methods offering basic interaction possible on different elements on a web page. With the core class ready all methods in page objects will refer to methods in it, this encapsulates tool specific activities largely to one core class, this makes the page objects free from tool/ library specific code.

This opens an opportunity of moving to a different tool/ library in case required with savings of test code, page objects, and Web Element repository. It also creates an opportunity of using multiple tools in core classes beneath Page Objects to get the automation job done.

4. Blueprint of the test automation harness



Takeaway: Relation between Test Tool abstraction, Page Objects, Web Element Repository, Test Cases, Test Suites, Web Driver, Test Engine, Browser…

Summary

The challenge of building test automation harness was not an easy one. What mattered most for us was standing up to the challenge of using a free / open source library to achieve our goals in low cost manner. Today we successfully execute automated top 10 user workflows and Build Acceptance Tests everyday and save more than 6 engineering hours every week on execution both test automation suite.

We are still on the journey of making improvements on the test automation harness and increasing the Return on Investment, the test automation harness is integrated with fitnesse (http://fitnesse.org/) acceptance testing framework and being used by a team of 18 engineers for carrying data driven testing on already automated workflows. We have come a long way in implementing an automation test harness for a very complex and dynamic web application from having absolutely nothing to begin with – along the process in addition to the explicit ROI the team has gained absolute experience and is now in a position to accept any test automation challenge with a “can do” confidence.

References

http://en.wikipedia.org/wiki/Factory_pattern
http://code.google.com/p/selenium - Google WebDriver now comes bundled with Seleniun 2.0a5 bundle
http://code.google.com/p/selenium/wiki/DesignPatterns
http://fitnesse.org – Acceptance Testing Framework

Assessing the Health of Your QA Organization

Michael L. Hoffman
capt.navar@gmail.com

Abstract

The aspiration of any software company or organization is the delivery of software products within defined goals of scope, and constraints of schedule and resources. The competitive nature of business requires that organizations live up to these goals, and work within these constraints, more effectively and efficiently over product lifecycle generations.

What constitutes improving over time for a software QA organization varies depending on perspective:

- For a tester it may mean more quickly differentiating between improper and expected behavior.
- For a customer support analyst it may mean less issues found in the field.
- For a test engineer it may mean optimizing test coverage based on risk assessment.
- For a quality engineer it may mean identifying issues in earlier lifecycle phases.
- For a QA manager it may mean making sure a quality/test team has the proper knowledge and tools in time to validate new technologies.
- For a QA director it may mean doing more with less.

The ability to consolidate so many perspectives into a comprehensive evaluation of improvement at the organization level is a complex challenge. A QA organization needs to regularly ask itself these questions:

How do we know we are doing the right thing?
How do we know we are improving over time?

Realizing quantitatively whether your software QA organization lives up to the dynamic needs of your business is not as simple as looking at defect trends over time. An appropriate evaluation involves various aspects of a software QA function, including: operational behaviors, talent, customers, and budgets.

This paper chronicles an approach taken by one software QA organization to evaluate its own health through establishing goals, benchmarking, defining organization-level metrics, and on-going self-assessment.

Biography

Michael L. Hoffman is a 20-year veteran of the software development/QA industry. He has filled a variety of engineering, management, and strategist roles at companies including Jeppesen Sanderson, Hewlett-Packard, Insight Distribution Software, Systematic Designs, Sharp Microelectronics, and GTE Government Systems. Michael's experience includes the areas of communications systems, factory control/integration, embedded systems, PC applications, and web-based solutions. Additionally, Michael teaches various software courses for Oregon Institute of Technology and is Chair of Oregon Tech's software engineering department industry advisory board.

1 Introduction

The success of delivering products and/or services from the perspective of a software QA organization is typically measured by looking at factors such as working within budgets, timely releases, and number/severity of field defects. Metrics around these factors tell how well the organization is doing, but do not indicate whether the organization is effectively aligned with the business.

A good team or organization can perform well, deliver on time, have low field/warranty costs, and still fail to meet goals; particularly if there is not clarity as to how organizational performance is being measured.

It's not matter of accountability. It's a matter of being accountable for the right thing.

Below is the experience that one software quality organization undertook to assess its delivery of value to the company and to ensure that it maintained that value over time.

2 The Challenge

Our Software Quality Assurance (SQA) organization was made up of seven geographically-separated teams that focused on testing and quality improvement of software for consumer electronic products.

Following a high-level company reorganization, SQA reported to a vice president who had not previously been responsible for a QA function. To become familiar with our organization, the vice president requested an overview presentation.

Nearly a week was spent by the SQA leadership team planning, preparing, and practicing our shtick; the resulting presentation was seamlessly delivered. Our goal of showing that we had a well-organized, cohesive SQA organization that consistently delivered on-time, in budget, products, with low field/warranty costs was accomplished.

Our audience, the vice president, accepted our assertions as to our success and did not doubt that there was high value in what we did. Following the presentation he asked us three simple questions:

> *How do you know you are doing the right thing?*
> *How do you know you are improving over time?*
> *How do you compare to similar organizations at our competitors?*

What he was really asking was for us to prove to him that what we were doing was what the business needed from us, that we were getting better and better at our role within the company, and that our approach was the best possible.

We were not prepared for, nor was the vice president expecting, an on-the-spot response to that question. We left the meeting with the task of determining the answers.

3 Determining the Approach

SQA management determined the current approach to metrics was accurate at showing whether we were successful in qualifying the readiness of products for the company, but they also conceded that providing this information was just not sufficient in showing whether we were delivering to the business' goals or that we were constantly improving our value to the company over time.

A cross-organization project team was formed with the goal of defining and driving a plan that would lead to a way we could assess SQA that answered the three key questions posed by our vice president.

The following approach was planned:

- Taking stock
- Benchmarking
- Understanding where we stand
- Defining measures
- A first assessment
- Setting thresholds and actions
- Continuing to focus on the right thing

4 Taking Stock

In today's work environment, virtual teams introduce a variety of factors that impact the effectiveness of an organization. Even people within the same reporting structure can have vastly different views of their organization, influenced by factors such as business unit history, target market segments, culture, and personal experience.

We realized very quickly the need to have a common realization of our own SQA organization before we could effectively benchmark with other companies. We developed a template to document our own self-assessment. In doing so, we also had a tool for sharing information with benchmarking partners and for documenting our assessments of other companies. Our assessments were organized into key categories:

Category	Description	Discussion Points
Organizational Model	How it impacts objectives, priority-setting, and decision-making.	Organization structure Management of outsourcing
Relationship to Customer	How SQA is assessed and held accountable for its responsibilities.	Customers' view of the quality organization Relationships with partner organizations Assessing satisfaction
Roles and Responsibilities	SQA roles, their respective responsibilities, and the partnerships with external teams.	Roles/responsibilities within SQA Assessing/tracking performance On-going individual development
Product Assessment	How product readiness is assessed.	Tracking the health of products Assessing product readiness Historical/predictive analysis
Process Assessment	How the effectiveness of quality/test processes is assessed.	Lifecycle processes Proactive quality processes Assessing process effectiveness Measuring the right things
Organizational Assessment	How the health of the quality organization is assessed and improved-upon.	Responsibility for organization health Assessing/measuring organization health Alignment of organization goals
Organization Performance	How the quality organization is measured by the company.	Measuring organization performance Effective metrics at an organizational level

Although much of the self-assessment was simply a matter of bringing together existing information, the exercise itself was valuable from a few perspectives. It provided us with:

- a realization of levels at which we already were doing self-assessments
- some obvious self-assessment gaps that, until this point, had never been recognized by our leadership team

- a trigger for conversation topics for our upcoming benchmarking efforts
- the basis for a set of information we planned to share with participating benchmarking partners

5 Benchmarking

The goal of benchmarking was to discuss the key categories with SQA organizations outside of our own.

Benchmarking was approached as a mutually beneficial opportunity. As part of the process of soliciting benchmarking participants, we offered an open, two-way conversation (any question we asked of them, we'd be willing to answer ourselves) and we committed to sharing our final benchmarking report.

In order to answer the question "how do we compare to our competitors?", it would have been ideal to benchmark with a competitor. That, however, certainly was not realistic. We did not want to share our business practices with competitors and we suspected they would feel the same way about sharing with us. Alternately, we chose to benchmark with companies that had similar target market segments (high volume consumer electronics), but who produced products that did not compete with our company.

Additionally, we arranged to talk with a couple of companies in vastly different industries so as to include a diverse set of perspectives. Two non-producers of consumer products were chosen: one that focused on the service/logistics industry and the other a major defense contractor. All participants were representatives from quality assurance teams.

Participating companies were provided with a general list of discussion topics (see above) based on the key categories, to set the expectation of discussion scope and to facilitate productive conversations. In a few cases, benchmarking participants reciprocated with additional questions that were incorporated into the benchmarking discussions.

Benchmarking was accomplished with half-day or full-day sessions based on time availability of participants. All sessions were conducted in person, except for one session that was conducted via video conference.

6 Understanding Where We Stand

Each company involved in the benchmarking effort was successful in qualifying their products for market readiness, but no two companies did so in the same way, nor did they use the same measures to assess their success or determine if they were improving over time. This was a result of a number of factors:

- The nature/complexity of the technology in their products
- Commitment of the company toward the QA function
- Maturity of their SQA organization in terms of quality practices
- Resource (people and/or budget) constraints

Consequently, realizing how we compared to the QA teams with which we benchmarked became a subjective activity. Since benchmarking discussions were focused on category-based questions, the categories were used as the comparison points. For each category, the relevant learnings that came out of the benchmarking with each company were discussed and voted upon by our benchmarking team. Using ourselves as a midpoint for a scale, the results provided a picture of our relative comparison to each company.

7 Defining Measures

From our benchmarking results, it was clear that other QA organizations were similarly strong in defining metrics to assess the readiness of their products and the efficiency of processes. Some of the metrics used included:

- Number of defects per lines of code
- Test effort (cost) per defect found
- Mean time to fix / validation
- Number of re-opened defects
- Ratio of automated vs. manual test executions
- Coverage analysis
- Percentage of test plan reviews
- Number of escapes
- Number of quality audits

Despite the varying approaches to assessing product readiness and ensuring minimal defect escapes, there was very limited, if any, focus on assessment of improvement over time. Although this was reassuring in the sense that what we were trying to do was, in fact, beyond the normal focus of self-assessment and effectiveness/efficiency improvement, it showed that there was no simple, easy-to-implement method to accomplish what we wanted.

To move forward from this point, we focused on our company's scorecard provided by senior management. This included major categories of:

- Financial
- Customer
- Operational Efficiency
- Employee

Whatever measures of assessment we used, they needed to convey a story of organization wellness relative to these four focus areas. It was decided that our highest level organization metrics would be broken down by these four areas.

7.1 Financial

This area was straight-forward … measures were dictated by our company's financial organization for the purpose of consistency across the company, and there were tools and processes in place to collect data and provide analytical summaries. All that would be involved for SQA would be to include the already-generated measures as our part of our organization's self-assessment, and develop appropriate action plans as needed.

7.2 Customer

There was consensus that a metric around defect escapes into the field for previously undiscovered issues would give us the ability to assess SQA's effectiveness at determining readiness of a software release. The same is true for issues found during customer beta releases, but this metric would require consideration that software versions typically get released for beta testing before full regression testing has been performed.

From the benchmarking effort, it was discovered that many QA organizations own the responsibility of qualifying product usability in their charter. In fact, it was not uncommon for testers or test engineers to submit defects for areas where usability specifications were not met, were vague, or were undefined. In

the end, however, it was decided not to include this metric in the assessment of SQA since in our case, the evaluation of customer experience was owned outside of SQA, by a customer satisfaction team. The purpose of our metrics effort was to assess our organization's health, and that such a metric would actually be measuring how well our product met our customers' needs and not a reflection our organization's effectiveness.

7.3 Operational Efficiency

This area focused on domain-specific assessment. In our case, the domain was software quality/test engineering. The metrics selected were broken into five categories:

Categories	A measure of our organization's ability to…	Metrics
Readiness	Meet delivery milestones.	Specification readiness Test case readiness Product readiness*
Program Fulfillment	Live up to the intent of the program.	# of escalations # of waivers
Effectiveness	Do the right thing.	Defect removal Spoilage Lifecycle testing
Efficiency	Use less resources over time.	Effort per defect Defect merit
Initiative Status	Make timely progress on improvement efforts.	Varied by initiative

** Although this metric is actually a measurement of the timely delivery by development teams, the ability for SQA to deliver on its commitments on-time to the business are highly dependent on this factor. Additionally, this milestone does reflect effective collaboration between SQA and Development, so it includes aspects of self-assessment.*

7.4 Employee

We learned from the benchmark discussions with other companies that assessing employees is an area where there was no consistency other than tools such as performance reviews. But what we wanted to achieve was the ability to answer these kinds of questions:

1 Do our teams have the right skill sets needed to work effectively?
2 Are we prepared for what the future will demand of our organization (technology, partner-team changes, business model, etc.)?
3 Are employees satisfied with their roles in the organization?

#1 and #2 are really the same question, just focusing on different timeframes. We quickly realized that although there are tools for assessing the readiness of an organization in terms of meeting the needs of the future, we did not have a good handle on what would be expected of us down the road. Our business was changing (i.e. new market segments), technology was evolving, and new working models were being established (i.e. engaging vendors). This required new skill sets, and we did not know what those skill sets would be. We determined that before we could assess whether we had a sufficient workforce that was progressively getting better, we needed to put some work into determining goals to measure against. A side project was created to focus on determining skill sets for each of our organization roles, consider each person's capabilities against the roles they were filling, and from that generate assessments of two factors:

- Skill levels vs. roles
- Capable resources vs. future need

Note: a valuable bi-product of going through this process was information for each manager as to the development opportunities for each of their direct reports.

For #3, assessing employee satisfaction, a survey approach was deemed most appropriate. A survey was developed to gather employees' feelings toward their understanding of SQA's goals, their role within SQA, their feeling of support, and opportunities available to them to develop in the ways they wanted.

8 A First Assessment

The goals of taking a first assessment were to:

- Gain an understanding of the lower-level metrics and/or data acquisition processes/tools that were necessary to generate our higher-level metrics.
- Acquire data that could be analyzed to understand the validity of our measurements so we could determine if we were, in fact, measuring what we intended.
- Create baseline metrics values to compare against as we moved forward.

It quickly became apparent that we did not have sufficient processes and tools for the collection of the raw data needed for some of our metrics. For example, the readiness metric required tracking of deliverables (specification documents, test cases, product functionality) relative to milestones. Although we already had good practices in place for determining whether a milestone deliverable was met, we had no means of accurately or consistently recording these activities. Maintaining a spreadsheet checklist for each product was easily instigated as a short-term solution. Ultimately, a more robust web-based tool that allowed for setting up and tracking of deliverables on a product-by-product basis across all teams within SQA would be put in place.

In some cases, there was no short-term solution for a metric, and a bigger effort was required to reach the point where we had a necessary foundation for our metrics. The spoilage effectiveness metric is such an example. One of the key components to this metric is classifying a defect's root cause. Although our defect tracking system supported assigning a root cause to a defect, this was not a required field; plus, there was great inconsistency when this field was used. It became clear that simply changing our tool to require an indication of root case would not result in reliable data for our metrics. To achieve this, we needed to instill a cultural change in our partner development teams to realize the intent of the field and to adopt a behavior of determining "why" the code change was necessary. Was it due to:

- An incorrect implementation of a design?
- The correct implementation of a flawed design?
- A problem with the requirement?
- …

Influencing our partner development teams to encourage investigation into the true root cause of each defect did not happen overnight. It required a behavioral shift that would mature over time. Additionally, the task of going through tens of thousands of previously-resolved defects that had been created over numerous years and dozens of product cycles was deemed an unrealistic endeavor. As a result, it would be a significant period of time, on the order of a few product lifecycle generations (about one year), before a set of data would exist that could yield meaningful results for this metric.

Consequently, some metrics would be a "work in progress" until such a time that processes or tools matured, behaviors changed, and raw data became available.

9 Setting Thresholds and Actions

For those areas where valid, reliable data was available and metrics could be calculated and assessed, the next step was to determine what constituted acceptable results.

Given the broad span of metrics we chose, normalizing measurement units across the different metrics categories was unrealistic. We determined that each metric needed to be assessed independently and then acceptable vs. non-acceptable results reported in a common manner across all categories. Additionally, simply reporting a quantitative value for a metric was not sufficient. The current trend for each metric also needed to be determined and reported to fully convey whether improvement was being realized or whether we were losing ground in particular areas.

To normalize the assessment of both quantitative and trend results across all metrics, we adopted a simple color-coded reporting scheme:

Color	Meaning	Action
Green	Current quantitative results are within desired thresholds and trending indicates they will remain that way for at least the next reporting cycle.	None needed
Yellow	Current results are good but trending shows we may miss our threshold in the next reporting cycle. OR Current results are not within the target threshold but trending indicates it will be within the threshold by next reporting cycle.	Minor corrective action needed
Red	Current results are far outside the target threshold. OR Current results are not within the target threshold and trending says it will get worse.	Major corrective action required

This approach gave us the ability to use different units and/or scales and to track trends differently for each metric, yet use a consistent reporting technique for communicating overall results.

An attempt was made when initial thresholds were set for each metric to define the corrective action that would be taken for yellow and red conditions. Over time, however, we realized that the corrective actions we initially defined were not always appropriate to address particular situations. Consequently, we changed to an approach of determining appropriate corrective actions if, and when, a yellow or red situation occurred. Actions could then be designed and employed to address the exact root cause.

We encountered situations where there was an inverse relationship between metrics. This was particularly true in the area of operational efficiency. For example, often a positive trend for an effectiveness metric would result in a negative trend in an efficiency metric. At first seeing such results was disconcerting, but after understanding how improvements in one area can cause inefficiencies in other aspects of our operation, we realized that although the metrics results were not where we wanted them in all cases, it was a true reflection of where SQA had the opportunity to improve in the future.

The outcome was a realization that a narrative around our metrics was just as valuable, if not more valuable, than the status of the metrics themselves. The work that went into understanding why the metrics were what they were and why they were trending in a certain direction yielded the most useful information of all, both to us and the recipients of our assessment reports. It gave us the ability to:

- understand if we were doing the right thing
- assess whether we were getting better over time
- realize our opportunities for improvement

… which is what this endeavor was all about: assessing the health of our SQA organization.

10 Continuing to Focus on the Right Thing

Determining the period of time for generating and assessing SQA organization metrics turned out to not be as simple of an effort as we anticipated. Some of our metrics were driven by product life-cycles (which varied greatly depending on the complexity of products), some by product release dates (i.e. peak selling seasons), some by financial milestones (i.e. fiscal quarters/years), etc. There was no one-size-fits-all period for assessment.

At the same time, selecting a fixed period (i.e. monthly) of collecting data, generating metrics, and doing assessments did not make sense either. A specific metric does not necessarily yield good results based on a periodic calendar date. For example, calculating the number of field escape defects for a product on the first of each month does not make sense while the product is in development. That kind of metric needs to be generated only after the product is released.

We chose to collect data and calculate each metric separately, based on the period for that metric which yielded a meaningful result. Assessments, however, were made based on a fixed period of time; we chose quarterly. Although not all metrics were updated each quarter, we did have the ability to report our organization assessment regularly. Plus, this ensured that what we reported accurately reflected the results for the most-recent time the metric was generated.

11 Summary

If there is one truth we learned as a result of this effort, it's that complacency in a QA organization leads to lower quality over time.

A company that produces a market-leading product will succumb to eventual competition if it does not continue to innovate and become more operationally efficient in how it brings products to market. The same is true with complacency in a software quality organization. Without continuous innovation in our approaches to quality, and on-going improvements across all facets of SQA, the quality of products will degrade over time due to the ever-increasing demands of the product's technology and feature set, and the ever-shrinking budget and time-to-market constraints.

Using an organization assessment approach such as the one described in this paper will help in determining if there is, in fact, improvement over time. However, we must continuously ask whether the improvements we are making are really what the business needs. So in addition to looking at our metrics to determine whether we are getting better, we also need to continuously ask whether we're measuring the right things.

Adopting an approach to regularly ask ourselves the fundamental question "Are we doing the right thing?" will help determine if the metrics accurately reflect what the organization needs to know as to how it effectively and efficiently it delivers to the needs of the company.

Software Quality Management System

Omar Alshathry
Software Technology Research Lab
De Montfort University
The Gateway, Leicester LE1 9BH, UK
shathry@dmu.ac.uk

Abstract

In software development projects, the investment of quality improvements needs to be optimized in a way that does not affect the cost and schedule aspects. However, as is currently practiced in the industry, software's artifacts are considered equal in their significance and risk to the software life cycle with respect to quality improvement activities. The investment in activities concerning the detection and removal of defects is distributed evenly on the software's artifacts without taking into consideration the risk and significance factors of such artifacts. Some software's modules hold risky and significant architectural components that need to be of a high quality and a low defect density. On the other hand, other modules do not require a similar level of quality. Defects originating from modules of high criticality may contribute to a project failure more so than less significant modules. In this paper, we propose a model that helps the project managers to optimize the investment given to the QA activities of their software on the basis of the risk associated with the development process.

Keywords: CosQ, Software Quality, Cost of Quality, Return on Investment, Quality Management

1: Introduction

A well known phrase among software quality practitioners is, "too little is a crime, but too much testing is a sin". This statement reflects the ongoing research problem in the Quality Assurance (QA) area when project managers, at the start of or before the testing phase, should make informed decisions to tune and control the triple constraints in a way to assure the success of their software projects **(Figure1**).

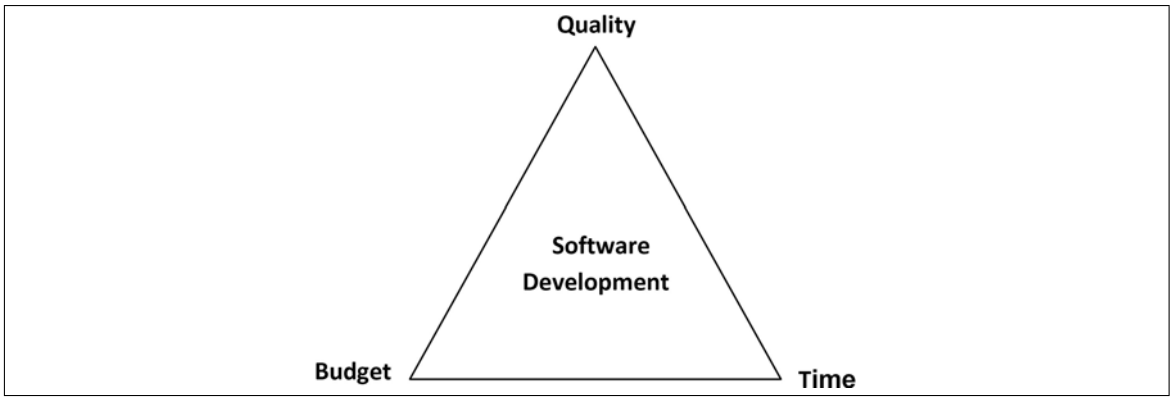


Figure 1: Software Triple Constraints

In some cases, schedule becomes the most important aspect when cost of delay outweighs the benefit of producing software of low defect density. In other cases, a limited budget assigned to the quality plan hampers the quality assurance team from covering software work products sufficiently to detect and remove defects, even those that are deemed to be of a high or moderate risk.

To tackle this issue, we propose a software quality management system that helps the project manager and QA practitioners to make informative decisions on their QA activities in terms of their cost effectiveness and to trade off alternatives of QA plans based on optimal solutions generated. We propose a regression-based generic model that categorizes each each phase's artifacts to different work products according to pre-defined risk levels so as to prioritize investment given to the quality assurance process.

The outline of our paper is as follows. In **Section 2**, we outline the cost of software quality (CoSQ) principal and the cost percentage that cost of software quality consumes out of the total software development budget. In **Section 3**, we outline our quality model informally. The formal descriptions of the model then follows in **Sections 4**. A literature review of similar models is discussed in **Section 5**. We conclude our paper with a critical review of our model in **Section 6**.

This paper is built upon our previous research on risk-based QA activities management [24][27].

2 Software Quality

2.1 Software Quality Perception

In our research, quality of software is intrinsically linked to the notion of software defects and defect density. Software defects can be defined based on different views and perspectives [15][2]. However, it is commonly agreed that a software defect is any flaw present in the software that prevents the normal software execution, causing software failure or nonconformance to software specifications [18]. We chose the defect density metric as our perception of quality and it is calculated as follows :

$$\text{Defect Density} = \frac{No.\ of\ defects\ in\ a\ code}{Size\ of\ the\ code}$$

2.2 Cost of Software Quality

The term cost of software quality *(CoSQ)* describes the trade-off between delivering high quality software and the cost associated with it [1]. It is a metric that helps software project managers to determine the accepted and affordable level of quality for their product.

It is generally accepted that during the software development process, the earlier a software defect is fixed and removed, the more effort and time is saved for the whole project [37] [32]. Software quality researchers estimated that software defect that cost less than $1 to be fixed during software coding phase may cost $100 if it propagated to the whole system and thousands of dollars if it passed to the operational field [38]. This is also supported by Humphrey[23] who showed that the rework cost of a defect found in the field is ten times greater than the cost to remove it during the system testing phase. The rationale behind this escalation in the cost of defects is that to fix a defect which was discovered relatively far from its origin entails not only the cost of fixing it but also the needed re-working[8][10] of other modules impacted by this defect in previous stages **(Figure2**).

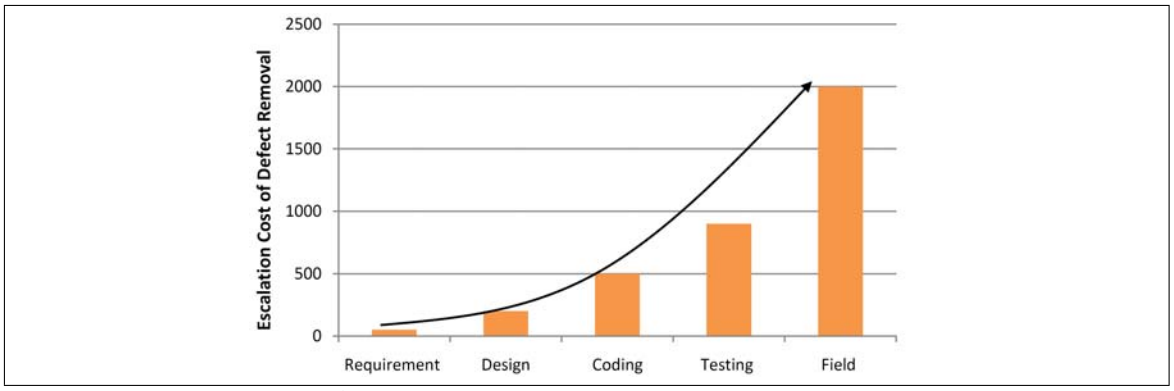


Figure 2: Escalation Cost in Software Defect

2.3 Software Quality and Profit

As shown in **Figure 2.3**, the relationship between software quality and profit is linear at the beginning, as implementing practices and techniques removes defects and improves software quality. However, this linear relationship reverses at one point during the software development process with a decrease in expected profit and a continuous increase in quality. On the other hand, some researches found that Pareto Law, known as the 80:20 rule, is applicable to software quality activities. It was found that 80% of software defects discovered in the system testing phase are related to 20% of the software modules[7][12]. Those two concepts of profit of software quality and the

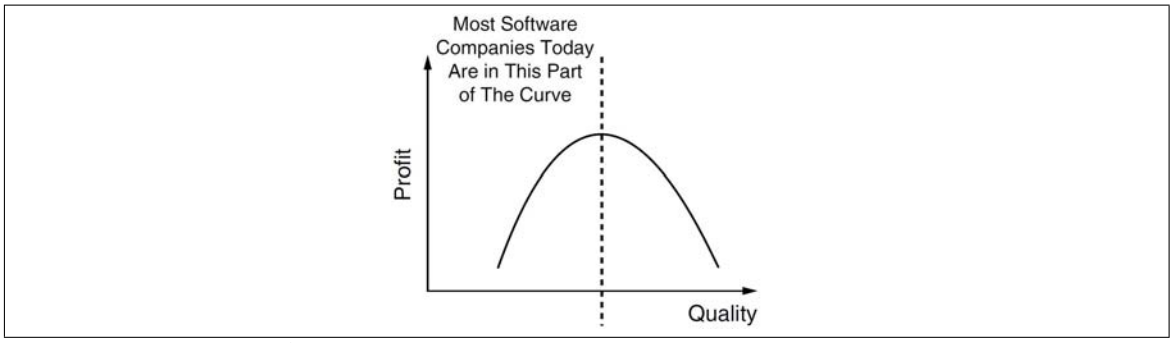


Figure 3: Relationship Between Quality and Profit [5]

Pareto Law are important in any software development project especially when it experiences a shortage in budget assigned to quality or time to release constraints. However, what are the criteria behind the decision of leaving work products uninspected to meet release time targets or re-using a module to save testing cost; how the project manager can handle such issues with minimal risk. In that context, there must be a mechanism that gives the project manager the ability to differentiate between high-risk modules that require more investment in QA activities and low-risk modules.

3: Quality System Model

Our approach to the optimal use of quality assurance practices and performing the required trade-off process between the software triple constraints is regression-based. In other words, we perform a regression analysis on a pool of QA data which was grouped and channelled according to our model specifications. The output of this analysis process will help determine the accurate effectiveness of

QA practices, their cost and time with respect to categorized software work products on the basis of the risk level associated with them. The application of our model conforms to any software development life cycle as long as it consists of phases. As shown in **Figure 4** which describes the work-flow steps of our model, each phase's deliverable will go through a categorization process to categorize each artifact to different work products.

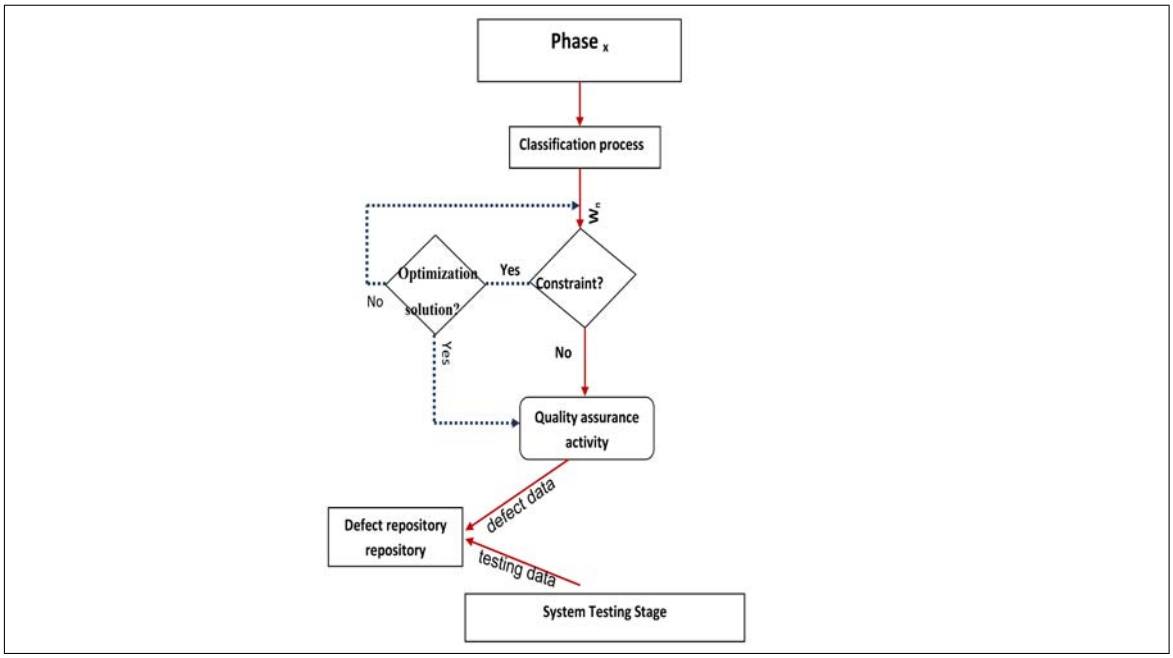


Figure 4: Work-flow of our Quality Model

The next step after the categorization process is the constraint check, which is a decision-based process conducted by either the system's project manager or the system's stakeholders who may put some items and conditions to be considered in the system to be built. Once constraints have been determined they go through the optimized solution determination process. In this process, constraints are grouped together and processed using a suitable optimization technique which will propose an optimal solution for QA plans. The optimal solutions found will be based on two or more of the software triple constraints: cost, quality and schedule. In case there is no optimal solution found, the constraints should be rediscussed between the software stakeholders and the project manager and redesigned for better manipulation of the optimization process. Once optimal solutions are found or in case there were no constraints determined, the QA team should start the QA process using suitable practices. Data of any QA activity in terms of the number of defects found, the duration of executing the QA activity, the average defect removal cost by the QA practice, etc., will be passed to the defect repository process where it is channeled and stored according to the categorized work products.

3.1: Software Models Categorization

The project deliverable of each phase in the SDLC can be broken down into work products. We assume that these work products can be assigned to a domain of specific type and risk categories. For a limited QA planning budget, the highest share of the budget should go to the most critical work products rather than being distributed evenly. For example, in a software requirement specification document requirements are typically ranked based on importance to the client. Here

more effort should be given to the verification of requirements that are important to the user than to those that are in the "waiting room"[11]. Having the risk levels determined, each work product of software development phases is categorized based on a defined set of risk levels derived from the consideration of the risk associated with the software development process. The categorization process is as follows:

Let C be a set of all categories that are used in the domain for which the software is being developed. Every work product can be placed in one category. Let W_x be the set of work products that are generated in phase x. The function $type : W_x \mapsto C$ then assigns for every work product a type category. Multiple categorization schemes can be modelled along these lines. Similarly let $esize : W_x \mapsto N$ be a function mapping from a work product to an estimated size in kLoC. Although for early work products other size measures, such as object or function points, are more suitable, it is well established to translate these into kLoC for the target language of the project [20]. The size $size_x$ of the output of phase$_x$ is then

$$size_x = \sum_{w \in W_x} esize(w)$$

The size of work-products of a specific type $c \in C$ is then:

$$size_{x,c} = \sum_{w \in W_{x,c}} esize(w)$$

where $W_{x,c} = \{w \in W_x | type(w) = c\}$.

The proportion of work products with that are in category c is then:

$$\alpha_{x,c} = \frac{size_{x,c}}{size_x}$$

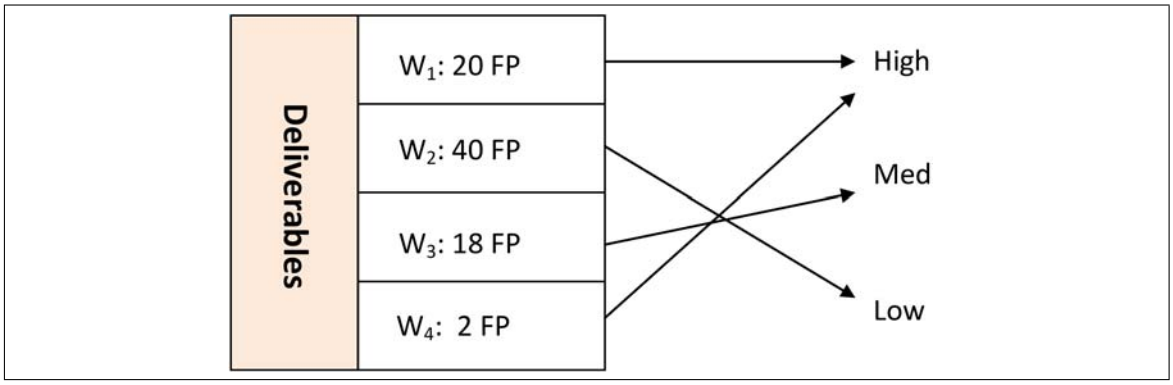


Figure 5: Phase Categorization Process

The categorization of a phase's deliverable work products depends on the project manager insight or the QA team consideration for the deliverable itself and for the type and prospective domain of use for the software. Some deliverable would be given a full weight value $\alpha_{x,high}$ of a high risk rating out of the total deliverable size $size_x$. Conversely, others will be given full $\alpha_{x,low}$ weight of low-risk rating. An example of a mapping from work products to risk categories is shown in **Figure 5**.

3.2: Categorization Scenarios

The process of categorizing software phases into work products is independent of the software size and the risk levels defined. That is, having a set of risk levels does mean that each phase deliverable need to be categorized according to these levels at once. The constraints experienced during software development process and the project manger insight are the main determinant factors for the way this categorization process is implemented. In the following subsections, we will show how this methodology is applied to the software requirement specifications by giving some scenarios of work products categorization.

3.3: 1st Scenario

As an example and as illustrated in **Figure 6**, it shows that a Software Requirements Specification (SRS) document is divided according to our risk-based levels (*high, medium, low*) into three work products of sizes 30%, 50%, 20%, which are inspected by QA practices P_1, P_2 and P_3 respectively. The result of this inspection process is the Defect Removal Efficiency (DRE) value for each QA practice with respect to the type of the work product it is applie to. The DRE value will be known to the QA team at the system testing phase when total defects originated from the work products uncovered.

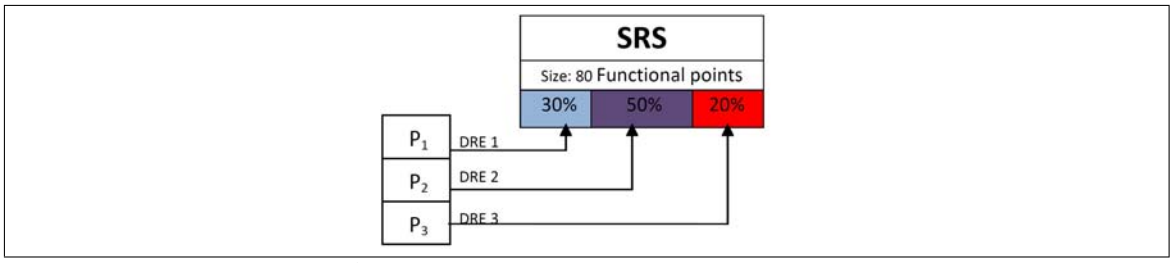


Figure 6: Work Product Categorization (a)

3.4 2nd Scenario

In another scenario, **(Figure 7**), the project manager may have no available budget to inspect all the artifacts or some of the QA practices may only be available for a limited period of time. Accordingly, the QA team manager chose to inspect the high and medium work products only, which is weighted at 50% of the total artifact with the coverage values of 30% and 20% for QA practices P_1 and P_2 respectively, and leave the left 50% of low-risk work product uninspected.

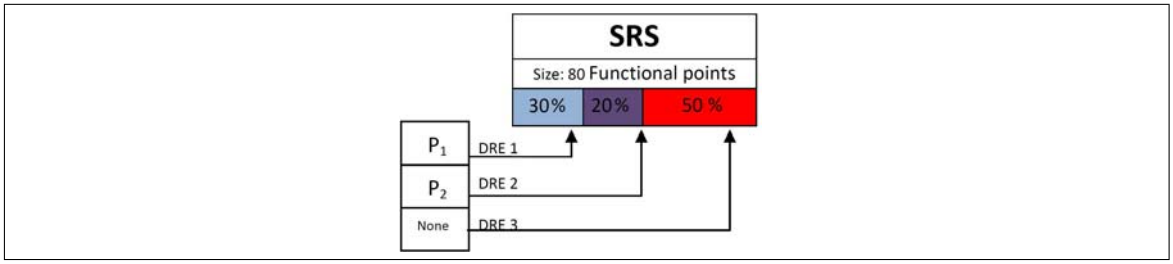


Figure 7: Work Product Categorization (b)

3.5 3rd Scenario

The Categorization process may continue to include the work product itself with respect to the coverage weight given to the QA practices chosen. For example, in **Figure 8** , the project manager may not give practice P_1 a full coverage on the work product of type high because P_1's availability stopped all of a sudden in the middle of the testing or due to other constraint. Accordingly, the project manager used practice P_3 to inspect the remaining part of the work product.

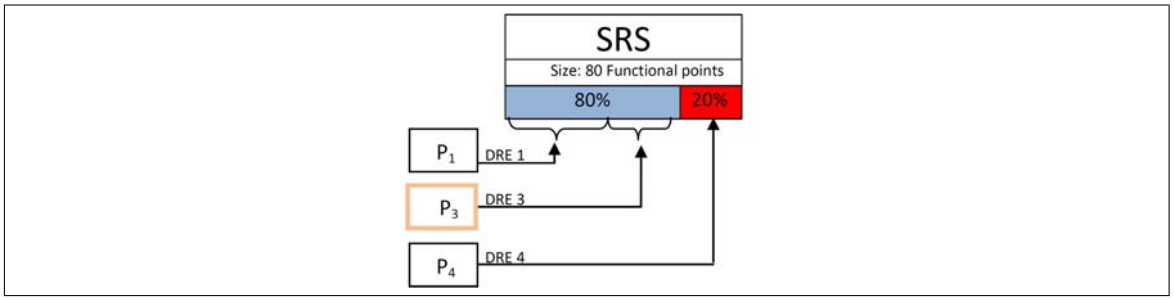


Figure 8: Work Product Categorization (d)

Those three different scenarios mentioned above occur repetitively in software development environment without exploiting them for future use. Software organizations should have a comprehensive and solid background on their QA practices efficiency for each phase and should also have a QA repository of previous projects.

3.6: Data Repository of QA activities

The application of this approach for many projects following the same software development process enables the QA team to build up a data repository of defects reports. In this section, we will show how data resulting from our model can be sorted and processed in a database so that it can be utilized for making decisions for future usage.

After the work products categorisation process, the QA team stores in the data repository all relative data related to each work product like its size out of the total phase size and its risk level **(Table 1)**.

Software Requirement Specification			
Project$_{id}$	Document size (FP)	Type of work product	Work product weight
1	Size$_1$	Type$_1$	$\alpha_1\%$
1	Size$_1$	Type$_2$	$\alpha_2\%$
1	Size$_1$	Type$_3$	$\alpha_3\%$
...	.....	.....	{100%}
2	Size$_2$	Type$_2$	$\alpha_2\%$
2	Size$_2$	Type$_3$	$\alpha_3\%$
3	Size$_3$	Type$_3$	$\alpha_3\%$
4	Size$_4$	Type$_4$	$\alpha_4\%$
5	Size$_5$	Type$_5$	$\alpha_5\%$
.	.	.	.
.	.	.	.
.	.	.	.
n	Size$_n$	Type$_n$	$\alpha_n\%$

Table 1. Work Products Input Table

Having completed the data input process, the data analysis process starts by grouping the dependent variables together for analyzing and determining the relationships between them. The mechanism on which our analysis process is based is the average values of variables and the regression analysis. Our average values determination process will analyze all QA data of past projects that are stored according to our model structure. This process will be applied on the following variables and database tables:

- **Each QA practice with respect to the phase and work product type.**
  The result of this table will determine the average DRE value for a specific QA practice with respect to the work product and to the phase it is applied to.

Phase	Work product	QA practice	$\bar{DRE}$

- **The execution time for each QA practice with respect to the phase and work product type.**
  The result of this analysis process is to determine the average time $\bar{time}$ value needed by QA practice p_1 to run on a work product w_x

Phase	Work product	QA practice	$\bar{time}$

- **Each QA defect removal cost with respect to the phase and work product type.**
  The result of this association will be:

Phase	Work product	QA practice	$\bar{cost}$

To calibrate our model we require a defect injection rate (I) that links the number of defects injected in every work product with its size. Given a positive correlation between the size of work products of a specific type and the number of defects originating from that work product, we can use linear regression to determine the injection rate Ic as the slope of number of defects originating from work products of that type against the size of the work products **(Figure 9)**.

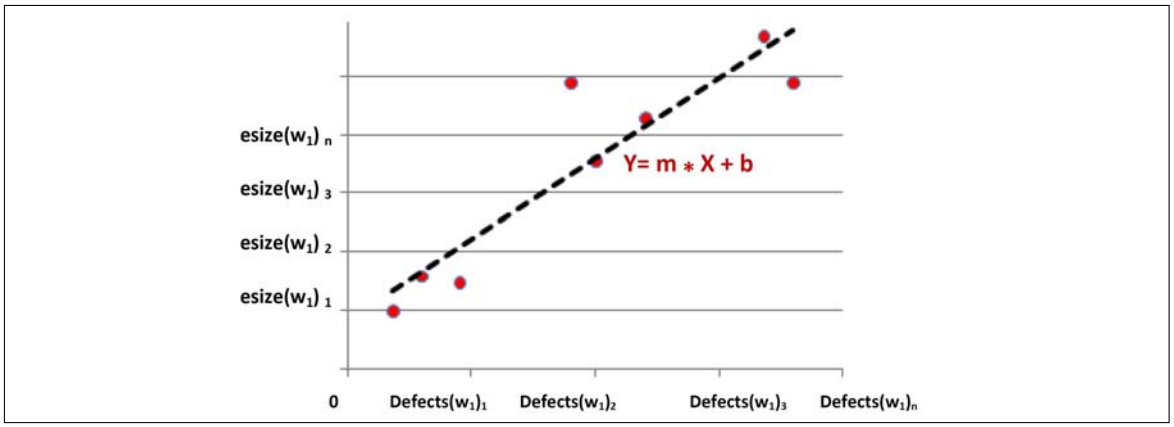


Figure 9: Proposed Regression Analysis

4 Formal Model

Following the SDLC approach being used by the software development organization, our software quality model consists of a sequence of phases: requirement, design and code which are referred to as (r, d, c) respectively. Moreover, each phase consists of a number of work products $w_1 \rightarrow w_n$. On the other hand, there are a number of QA practices which are specifically used and responsible for the defect detection and verification activities within each phase **(Figure10)**.

4.1 Model cost aspects

Generally, the formal model of our system consists of three main components as follows:

1. **Number of Defects (I)**
   This component refers to the estimated number of defects found by applying a specific QA practice to a single work product (w) or to a whole phase in general. The number of defects component of our model will be responsible for determining the DRE value with respect to the QA practice used.
2. **Execution Cost and Effort**
   The execution cost is the cost of performing the QA process using a specific QA practice. As we aim in this research to optimize the cost of QA investment, we need to accurately quantify this value to reduce the waste in effort introduced by any QA activity. In our model, the cost of execution is divided into two blocks:
   - Cost to run the QA practice.
   - Cost to remove defects discovered.

   The reason for this division is that during a QA activity, which comprises defect detection and removal, the defects found or some of them may not get removed during the QA activity even though they were discovered. the QA team may prefer to remove only part of the found defects due to unexpected constraints or because of the fact that the defects found are not necessary to be fixed.
3. **Cost of Escaped Defects**

   In this component, we quantify the impact of defects escaped from the main development phases of the SDLC: Requirement, Design, Coding, etc., with respect to the system testing

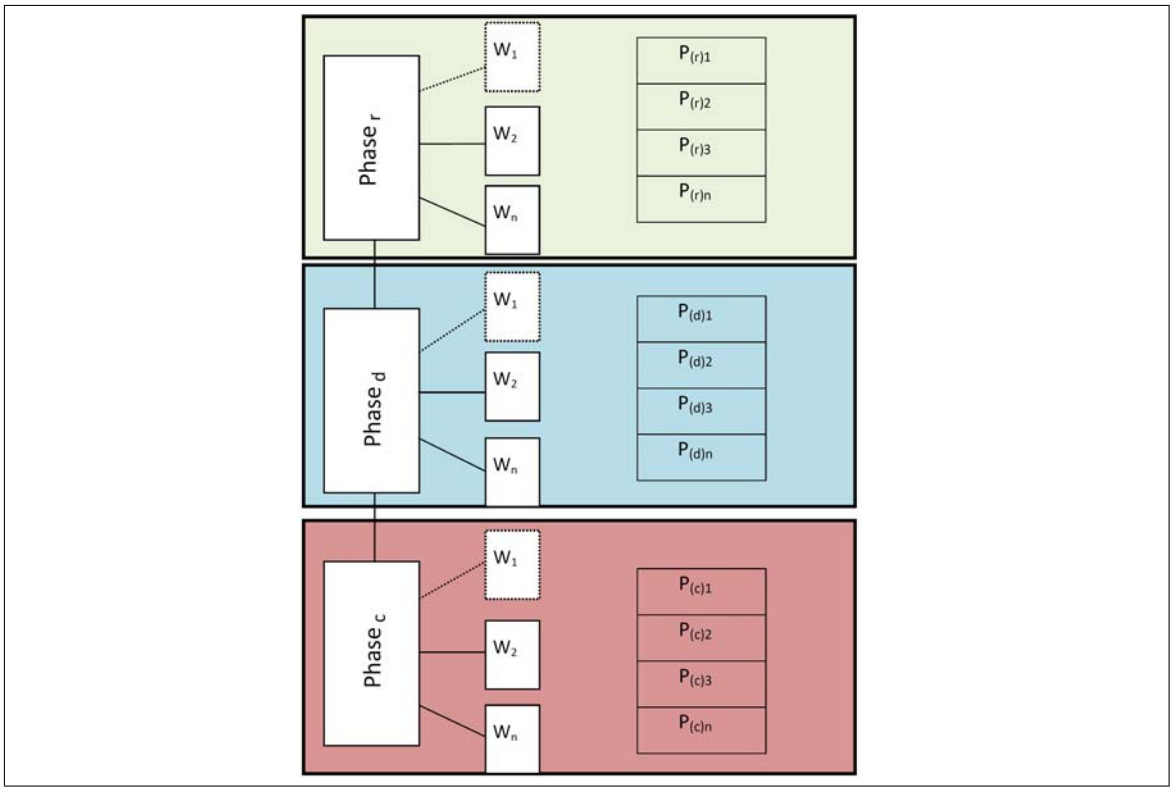


Figure 10: QA Practices and Phases Association

stage. An overview of this association is depicted in **Figure 11**.

The mechanism of our model works by associating QA activities carried out during the development activities of the SDLC's phases with the system testing phase, where the main testing activities begin. This association mechanism should quantify the impact of all escaped defects in terms of the estimated cost associated with their defect removal activities at the system testing phase.

As a result of this process, we will be able to measure the estimated cost introduced by a defect escaped from a work product of a specific risk level and discovered in the system testing phase. We refer to this value in our model as the escalation factor of a defect ($C^{escaped}$).

- P_x denotes the set of all QA practices that can be applied in $Phase_X$.
  Let $p \in P_x$
  Let $w \in W_x$
- Let I_w be the estimated injection rate of defects per kLOC in w of $phase_X$.
- Let β_p refers to the coverage weight of a practice p during a QA activity.
- Let $C^{escaped}$ refers to the escalation factor of an escaped defect with respect to the system testing phase.

1. Estimated Number of Defects Injected

$$eD_w = I_w * esize(w) \tag{1}$$

2. Estimated Number of Found Defects

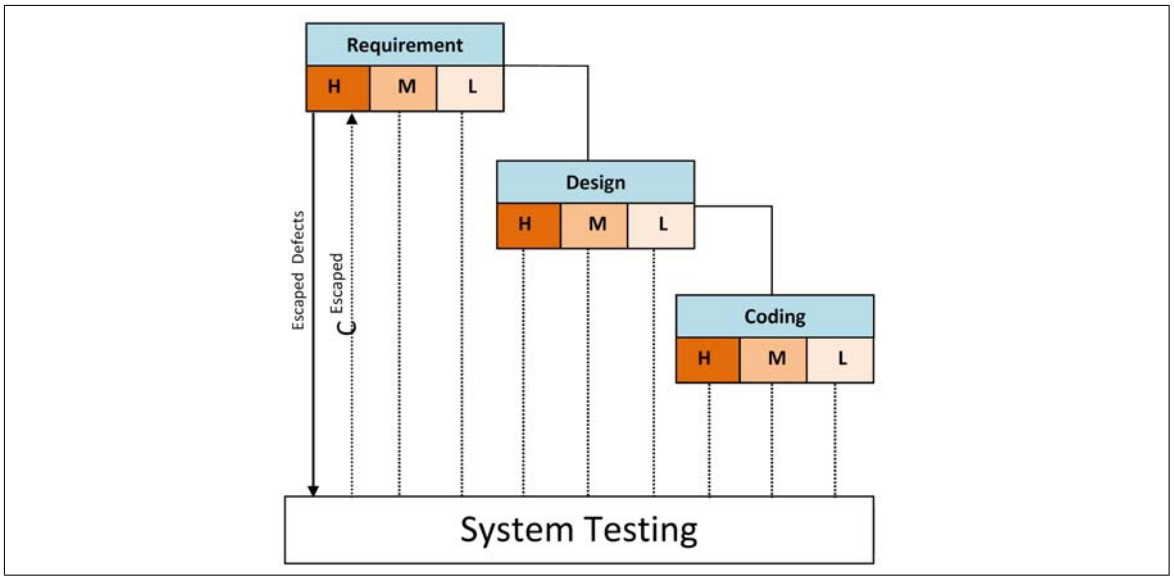


Figure 11: Cost of Escaped Defect

$$N_p^{Found} = \beta_p * eD_w * DRE_p \tag{2}$$

3. Estimated Number of Defects Escaped

$$N_p^{Escaped} = \beta_p * eD_w * (1 - DRE_p) \tag{3}$$

4.2 Execution Time and Effort

4.2.1 Execution Time

The execution time is defined in our model as th time required to run a QA practice on a software's artifact in order to detect and remove its defects. As we mentioned earlier, we need to quantify the cost of running the QA practice as a separate cost from the overall defects removal cost as in many cases software quality practitioners may find defects but not remove them.

- Let t_p be the average execution time of applying a QA practice $p \in$ P on a specific work product $w \in$ W.
- Let size$_w$ be the size of a work product measured based on the artifact's unit of measurement.

Execution time for a single QA practice p

$$Ext_p = \beta_p * size_w * t_p \tag{4}$$

4.2.2 Execution Effort

- Let C^{escaped} be the escalation factor of a defect removed at the system testing phase.
- Let Lr be the labour rate measured in any unit of money.

1. Execution cost for a single QA practice p

$$Exc = Lr * Ext_p \tag{5}$$

2. Cost of Defect Removal Rc

$$Rc_p = \beta_p * eD_w * DRE_p * C_p^{removal} * Lr \tag{6}$$

3. Cost of Escaped Defect Esc

$$Es_p = \beta_p * eD_w * (1 - DRE_p) * C_p^{escaped} * Lr \tag{7}$$

4.3 Saved Cost of a QA Activity

During the process of applying a QA practice, QA practitioners usually do not quantify a major important value that has a great influence in evaluating the current QA plan. This value is defined in our model as the saved cost which refers to the saving in cost of applying a QA practice which is expected to pay off later in the system testing phase.

$$Sc_p = \beta_p * eD_w * DRE_p * C_p^{escaped} \tag{8}$$

4.4 Combination of QA Practices

In some cases and for software artifacts of a high significance, the project manager may like to apply more than one QA practice at once in order to reduce the defects injection rate in later phases.

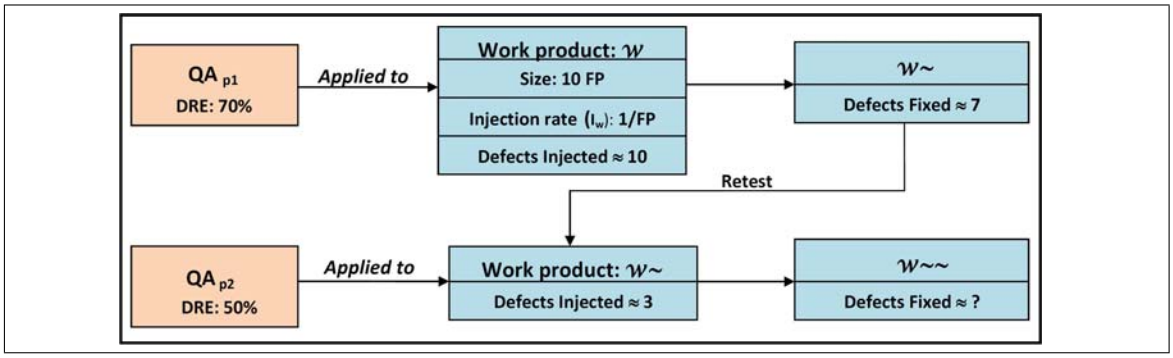


Figure 12: Combining QA practices

In order to clarify the notion of applying more than one QA practice, we give an example as shown in **Figure 12** that depicts a possible defect detection and removal scenario. As an example, the testing team chose to apply QA practice p_1 to a specific work product w that has an estimated injection rate (I_w) of 1/FP, that is, 1 defect is injected into each functional point of the work product. Assuming that the size of the work product is 10 FP, that means the work product is estimated to be injected with 10 software defects. The QA practice p_1 is known to have a defect removal efficiency of DRE = 70% on the work product w relying on data of past projects retreived from the model's repository. The estimated result of this activity was a new tested work product $\bar{w}$ which had approximately 7 defects that were fixed out of the original injected 10 defects.

70% * 10 = 7 defects found.

On the other hand, 3 defects are still injected in the work product $\bar{w}$ which will propagate to the later phases.

(1-70%) * 10 = 3 defects escaped.

In order to reduce the impact of these 3 escaped defects, the testing team decided to retest the same work product $\bar{w}$ using another QA practice p_2 which has a defect removal efficiency value DRE = 50% with respect to such work product type. However, The defect injection rate of work product $\bar{w}$ needs to be redefined according to the result of the first QA activity. We assume that the new injection rate of work product $\bar{w}$ is 0.3 FP as follows:

$$I_{\bar{w}} = \frac{Defect\ injected \approx 3}{Size : 10\ FP} = 0.3$$

Based on the values of $I_{\bar{w}}$ and the DRE of the QA practice $p_2 = 50\%$, the testing team would expect that the verification process would be estimated to reveal half of the injected defects. However, this optimisitic view of the second re-testing process may not always be correct; re-testing the work product with the QA practice p_2 may find no defects despite its defect removal efficiency value of DRE = 50%. The reason behind that is that in some cases the type of defects found by practice p_1 are the same defects found by practice p_2; therfore, applying practice p_2 on a work product which was already tested by practice p_1 may consume effort and time without tangible benefits.

This potential scenario is well-thought about in our research and we tried to tackle it by devising a new variable to calculate the probability that a QA practice p_2 will find defects different to those defects found by the preceding QA practice p_1. We call this variable λ. An overview of this variable is depicted in **Figure 13**.

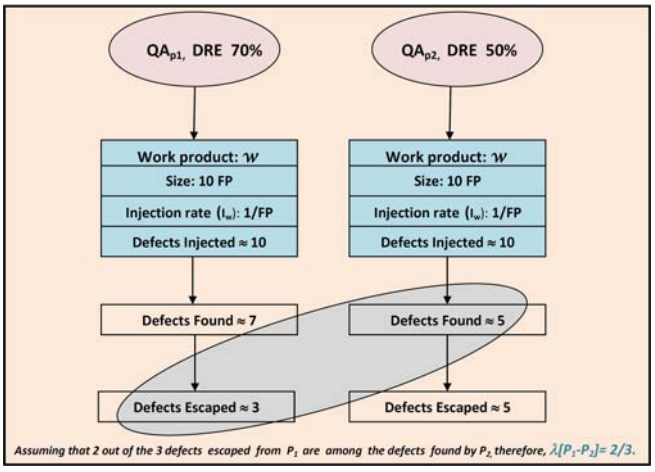


Figure 13: Lambda Variable Association

Based on **Figure 13**, the combination variable of applying the practice p_2 as a successor of p_1 is $\lambda = 2/3$. That is, QA practice QA_2 is able to uncover 2/3(two-thirds) of the escaped defects from practice QA_1.

By utilising the λ variable, the number of found defects after applying two QA practices sequentially is calculated by extending **Equation 2** as follows:

$$N_w^{Found} = \beta_{p_1} * eD_w * DRE_{p1} + \beta_{p_2} * eD_w * (1 - DRE_{p1}) * \lambda_{p1-p2}. \quad (9)$$

$p_1, p_2 \in P$

4.5 Return on investment

Defect detection and removal activities are considered to be an investment especially for profit-based software development organizations. This investment needs to be well-evaluated and studied to determine its positive and negative implications on the development process. The overall cost of any software development project is equal to the cost of the development activities and the cost of quality assurance practices implemented during the software life cycle.

Total development cost= Production development(COCOMO)+ Cost of quality

As our focus in this research is mainly on the second aspect of cost, which is the cost of software quality $CoSQ$, we will show how to evaluate the $CoSQ$ in a feasible way by using a well-known business measure, return on investment ROI. In our model, we express the ROI measure according to a value to cost ratio as shown in the following:

$$ROI = \frac{Value}{Cost} \tag{10}$$

Value : equals the saving in costs resulting from fixing defects early in their phases of origin.

Cost : equals the effort of both executing the QA practice and fixing defects found.

First of all, we recall the functions that we used in our model and constitute our model components. These functions are :

- Execution cost (Exc)
- Removal cost (Rc)
- Escaped cost (Esc)
- Saved cost (Sc)

In order to calculate the numerator for our ROI equation, (Value), we use the following expression :

$$\text{Value} = Sc - (Exc + Esc + Rc)$$

For each QA p applied to a piece of an artifact the estimated value is identified by subtracting the estimated saved cost from the other three aspects of execution effort: execution cost, defect removal cost and the escaped cost. The denominator in our ROI equation (Cost) can be defined as the sum of the three aspects of execution effort:

$$\text{Cost} = Exc + Esc + Rc.$$

Accordingly, substituting the two values in our ROI equation yields :

$$ROI = \frac{Sc - (Exc + Esc + Rc)}{Exc + Esc + Rc} \tag{11}$$

By generalizing the previous equation to the whole work product w, we can calculate the relative ROI of all QA plans applied to work product w by summing all variables as shown below:

$$ROI_w = \frac{\sum_{p \in P} Sc_p - (\sum_{p \in P} Exc_p + Rc_p + Es_p)}{\sum_{p \in P} Exc_p + Rc_p + Esc_p} \tag{12}$$

5: Related Work

Work related to our research includes similar models that work as a decision-based system to help project managers control the cost, schedule and quality of their software projects. One of the most well known cost and quality estimation model is the COnstructive QUALity MOdel (COQUALMO) [13] which is an extension to the COnstructive COst MOdel (COCOMO) [20]. It consists of two sub-models: 1, defect introduction and 2, defect removal models which in turn integrate into COCOMO to help determine the cost of defects removal using the three QA practices: *Automated analysis*, *Reviews* and *Testing*. However, In COQUALMO the effort to fix a defect introduced and removed by each of the three practices is not quantified directly by the model, it is calculated as a percentage of the total estimated effort by COCOMO model [6]. Also, along with the COCOMO, COQUALMO are calibrated based on data manipulation of many previous software projects which may incur difficulties for an organization to tailor any of those models to itself. Moreover, with COQUALMO there are only three applicable practices without taking into account the diversity of other techniques and the variations in their efficiency.

Capture recapture models is used in software engineering to predict the total number of defect in an artifact based on a sample taken from a population of defects [30][25]. However, the accuracy of defect estimations given by capture and recapture models is not stable and influenced by different factors like the type of QA practice [28], number of people involved [26] etc.

With regards to software QA practices and their defect detection and removal efficiency, Juristo et al [34] designed a classification schema for QA practices within software development life cycle upon a 25 years of testing experience. However, they concluded that there is a little knowledge related to QA practices and their applicable domain.

Gou et al [33] [16], analyzed data related to QA activities of historical projects and found that based on these data industrial baselines can be established to estimate defect removal and detection efficiency of current and future QA activities.

6: Conclusion

This paper is a part of ongoing research on a holistic model for software QA activities management. We proposed an approach that optimizes software investment assigned to QA processes by defining generic QA model whereby critical components within the software to be developed are given more consideration and priority .

The mechanism that our model relies on is that each phase deliverable is categorized into different work products according to predetermined risk rating levels introduced by software development organization in a way that high effective practices be applied to high risk rating level work products. We also proposed an adjustment to the defect containment matrix so as to make it able to determine the DRE for any risk rating level and accordingly the efficiency of each practice assigned to it. We presented set of mathematical equations that supports the theoretical model presented and to help determine the interaction and communication of our model aspects. Our QA model can be utilized as decision based QA system that helps software project manager make informative estimate on the consequences of daily based decisions regarding QA processes.

The aim behind this project research on which this paper is based is to come up with a holistic model to deal with software triple constraints schedule, cost and quality. There are many variables that were overlooked in this paper in order to make the proposed approach less complex.

Future research will include integrating the time consideration into the model so as to enable

triple constraints trade-off process. Ongoing contacts are being made with leading software development organizations to pilot the model for their software QA activities.

References

[1] Krasner.H, *Using the Cost of Quality Approach for Software*. CrossTalk. The Journal of Defense Software Engineering, pp11(11), 1998.

[2] Robert B., *Practical Software Metrics for Project Management and Process Improvement*, Englewood Cliffs, NJ: Prentice Hall, 1992

[3] Boehm. B and Huang.L. Value-based software engineering: A case study. Computer, pages 33-41, March 2003.

[4] Price Waterhouse, *Software Quality Standards: The Costs and Benefits*, A review for the Department of Trade and Industry. London: Price Waterhouse Management Consultants,1988.

[5] El Emam.K, *The ROI from Software Quality*, Boca Raton, Florida: Auerbach Publications, 2005.

[6] WANG,Q., GOU,L., JIANG,N., CHE,M., ZHANG,R., YANG,Y., LI,M. *An Empirical Study on Establishing Quantitative Management Model for Testing Process*, JCSP 2007, Lecture Notes on Computer Science 4470, 2007.

[7] Boehm.B, *Industrial software metrics top 10 list*, IEEE Software, September, pages 84-85, 1987.

[8] Jones.C,*Applied Software Measurement*, 3rd edition; McGraw-Hill, New York: 2008.

[9] Sower.V, Quarles.R, Cooper.S, *Cost of Quality:Distribution and Quality System Maturity: An Exploratory Study*, ASQ's 56th Annual Quality Congress Proceedings, ASQ, May, 2002.

[10] Fairley.R ,Willshire.M. , *Iterative Rework: The Good, the Bad, and the Ugly*. IEEE Computer 38 (9): 34-41, 2005

[11] ROBERTSON.S & ROBERTSON.R, *Mastering the Requirements Process*. Harlow, UK, 2006.

[12] LiGuo.H , Boehm.B, *How Much Software Quality Investment Is Enough: A Value-Based Approach*, IEEE SOFTWARE, 2006.

[13] Chulani.S, Boehm.B, *Modeling Software Defect Introduction Removal: COQUALMO (COnstructive QUALity MOdel)*, USC-CSE-99-510, The Center for software Engineering, University of Southern California, Los Angeles, CA, 1999.

[14] Votta.L, *Does Every Inspection Need a Meeting*, 1993.

[15] Florac, William, A., *Software Quality Measurement: A Framework for Counting Problems and Defects* , CMU/SEI-92-TR-22, Software Engineering Institute -Carnegie Mellon University, Pittsburgh, Pennsylvania, September 1992.

[16] Gou.L, Wang.Q, Yuan.J, Yang.Y, Li.M, Jiang.N, *Quantitative defects management in iterative development with BiDefect*, Software Process: Improvement and Practice, 14,4, pp.227-241, 2008.

[17] Chrissis. M.B, Konard. M, Shrum.S, *CMMI guidelines for process integration and product improvement*. Addison-Wesley,NJ, 2003.

[18] Boehm. B. and Basili, V. *Software Defect Reduction Top 10 List*, IEEE Computer, Vol. 34, No. 1, January 2001.

[19] Henderson.C, *Managing software defects: defect analysis and traceability*, ACM SIGSOFT Software Engineering Notes, 2008.

[20] Boehm B. W., *Software Engineering Economics*, Prentice-Hall, 1981.

[21] Chillarege.R, Bhandari.I.S, Chaar.J.K, Halliday.M.J, Moebus.D.S, Ray.B.K, and Wong.Y. Orthogonal Defect Classification-A Concept for In- Process Measurements, IEEE Transactions on Software Engineering, vol. 18, pp. 943-956. 1992.

[22] NASA., *NASA procedures and guidelines for mishap Reporting, Investigating and Record keeping*, Safety and Risk management Division, NASA Headquarters, USA, 2000.

[23] Humphrey.W.S, *A Discipline for Software Engineering*, Addison-Wesley Publishing Company,Massachusetts, 1995.

[24] Alshathry.O, Helge.J, Hussein.Z, Abdullah.A, *Quantitative Quality Assurance Approach*, niss, pp.405-408, 2009 International Conference on New Trends in Information and Service Science, 2009.

[25] Walia.G, Carver.J, *Evaluation of capture-recapture models for estimating the abundance of naturally-occurring defects*,Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement, pp.158-167, 2008.

[26] El Emam.K, Laitenberger.O, *Evaluating Capture-Recapture Models with Two Inspectors*, IEEE Trans. Software Eng., vol. 27, no. 9, pp. 851-864, 2001.

[27] Alshathry.O, janicke.H, *Optimizing Software Quality Assurance*, COMPSAC'10, The 34nd Annual IEEE International Computer Software and Applications Conference, 2010.

[28] ISODA.S, *A criticism on the capture-and-recapture method for software reliability assurance*. Journal of Systems and Software, 43, pp.3-10, 1998.

[29] L LAZIc, A KOLAcINAC, D AVDIC The Software Quality Economics Model for Software Project Optimization, World Scientific and Engineering Academy and Society (WSEAS) Stevens Point, Wisconsin, USA, 2009.

[30] Briand.L, El Emam.K, Freimut.B, *Comparison and Integration of Capture-Recapture Models and the Detection Profile Method*, Procs. Ninth International Conference on Software Reliability Engineering, Paderborn, Germany, pp. 32-41, 1998.

[31] Tsai.W. H., *Quality Cost Measurement Under Activity-Based Costing*, International Journal of Quality & Reliability Management, vol. 15, pp. 719-752, 1998.

[32] Capres.j, *Estimating Software Costs: bringing realism to estimatingý*, 2nd edition. McGraw-Hill, New York, 2007.

[33] Gou.L, Wang.Q, Yuan.J, Yang.Y, Li.M, Jiang.N, *Quantitatively managing defects for iterative projects: An industrial experience report in China*, Heidelberg, D-69121, Germany: Springer Verlag,pp.369-380, 2008.

[34] Juristo.N, Moreno.A, Vegas.S, *Reviewing 25 Years of Testing Technique Experiments*, Empirical Software Eng.,vol. 9, nos. 1Ű2, pp. 7Ű44, 2004.

[35] *Driving Quality Throughout the Software Delivery Lifecycle*, June 2007, Borland software corp. www.borland.com

[36] Wood et al, *Comparing and Combining Software Defect Detection Techniques: A Replicated Empirical Study*, Proceedings of the 6th European Software Engineering Conference, Zurich, Switzerland, pp. 262-277, 1997.

[37] Boehm.B,Valerdi.R, *The ROI of Systems Engineering: Some Quantitative Results*, eq-

uity,IEEE International Conference on Exploring Quantifiable IT Yields, pp.79-86, 2007.

[38] *Building a Better Bug Trap*, The economist , June19,2003,http://www.economist.com/science/tq/displayStory.cfm?Story_id=1841081 [accessed on] 16/oct/2009.

[39] Frost.A, Campo.M *Advancing Defect Containment to Quantitative Defect Management*, CrossTalk Ű The Journal of Defense Software Engineering 12(20), 24Ű28, 2007

Improvement Processes that Create High Quality Embedded Software

Jay Abraham
jabraham@mathworks.com

Marc Lalo
mlalo@mathworks.fr

Scott Runstrom
srunstro@mathworks.com

Abstract

The development of embedded software encompasses a wide range of best practices and development methodologies. For quality-critical projects intended for highly reliable applications, delivery of high quality software is an absolute requirement. In these situations, development and test teams must complete code reviews, perform unit and regression test, and testing on the target system. But is it enough? What if a critical defect escapes to software deployment and then to production? Formal methods based mathematical techniques may alleviate some of the doubt. Application of formal methods based code verification may provide precision in guiding software engineering teams to know which parts of code will not fail and isolate those aspects of code that will fail or most likely to fail. This paper will discuss the practical application of these techniques for the verification of software. As part of the application of this improvement process, the paper will explore how these techniques enable the creation of high quality embedded software.

Biography

Jay Abraham is currently Product Manager at the MathWorks. Jay began his career as a microprocessor designer at IBM followed by engineering and design positions at hardware, software tools, and embedded operating systems companies such as Magma Design Automation and Wind River Systems. Jay has a MS in Computer Engineering from Syracuse University and a BS in Electrical Engineering from Boston University.

Marc Lalo is currently Product Manager at the MathWorks with responsibility for Polyspace code verification products. He has been working in the embedded software verification and safety standards field for more than 10 years. Marc is a graduate of Ecole nationale supérieure des Télécommunications.

Scott Runstrom is currently a Principal Technical Writer at the MathWorks. Scott has worked in the technical communications profession for 20 years. His previous experience includes Applied Biosystems, Waters Corporation, and Stratus Computer Inc. Scott has an M.S. in Professional Communications from Clark University, and a B.S. in Technical Writing (with a background in mechanical engineering) from Worcester Polytechnic Institute.

1. Introduction to software quality and verification processes

The goal of most software development teams is to maximize both quality and productivity. However, when developing software, there are always three related variables: cost, quality, and time. Generally, the criticality of the application determines the balance between these three variables. With classical testing processes, development teams generally try to achieve their quality model by testing all modules in an application until each meets the required quality level. Unfortunately, this process often ends before quality objectives are met, because the available time or budget has been exhausted. To achieve maximum quality and productivity, however, one cannot simply verify and test code at the end of the development process. It must be integrated into the verification process, in a way that respects time and cost restrictions.

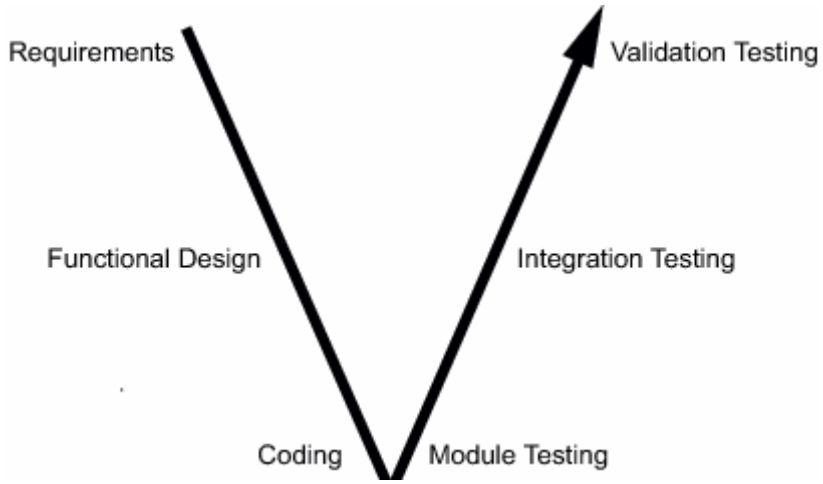


Figure 1 – Software Verification and Validation

The verification and validation process (often referred to as V&V) of complex embedded systems, consists of determining that the software requirements are implemented correctly and completely and are traceable to the system requirements. The main objective of the V&V process is to completely analyze and test the software during the development and test phase to assure that the software performs its intended functions correctly, to make sure that it performs no unintended operations, and provide information about its quality and reliability. Software V&V also determines how well the software meets its technical requirements and its safety, security, and reliability objectives relative to the system. The V&V tasks analyze, review, demonstrate or test all software development outputs (1). This process is often described with a V diagram as shown in Figure 1.

Specific classes of software defects that can be introduced in the coding phase of the V&V process are coding errors, run-time errors (software errors considered as latent faults) and design errors. The faults may exist in code, but unless very specific tests are run under particular conditions, the faults may not be realized in the system. Therefore, the code may appear to function normally, but may result in unexpected system failures. Software with these types of defects cannot be considered robust. A few causes of run-time errors are given below (this list is not exhaustive):

1. Non-initialized data – If variables are not initialized, they may be set to an unknown value.
2. Out of bounds array access – An out of bounds array access occurs when data is written or read beyond the boundary of allocated memory.
3. Null pointer dereference – A Null pointer dereference occurs when attempting to reference memory with a pointer that is NULL. Any dereference of that pointer leads to a crash.

4. Incorrect computation – This error is caused by an arithmetic error due to an overflow, underflow, divide by zero, or when taking a square root of a negative number.
5. Concurrent access to shared data – This error is caused by two or more variables across different threads try to access the same memory location.
6. Illegal type conversions – Illegal type conversions may result in corruption of data.
7. Dead code – Although dead code (i.e. code that will never execute) may not directly cause a run-time failure, it may be important to understand why. Note that DO-178B prohibits the existence of dead code.
8. Non-terminating loops – These errors are caused by incorrect guard conditions on program loop operations (e.g. for, while, etc.) and may result in a system hangs or halts.

The next sections examine how various techniques can be used to minimize or eliminate these errors in software and how these techniques can be incorporated into a V&V process to create high quality embedded software.

2. Static analysis

Static analysis or verification is a broad term, and is applicable to any tool which derives dynamic properties of a program without actually executing it. However, most static analysis or verification tools only verify the complexity of the software, in a search for constructs which may be potentially dangerous. As described in (2), these tools can certainly find some errors in code, however they may miss errors which strongly depends on dataflow (like run-time errors or design errors). Because these tools do not exhaustively analyze all behaviors these tools are considered unsound (miss errors) and therefore produce false negatives. Decreasing the probability of false negatives will increase the probability of false positives. i.e. falsely identifying a defect that is not real. The use of static analysis can provide a certain amount of automation in the verification process, but this advantage must be weighed carefully against the capability of these tools to generate false negatives and thus miss errors in the process.

3. Dynamic testing

Dynamic testing verifies the execution flow of software; e.g. decision paths, inputs and outputs. Wagner describes the methodology and explains the application of this testing philosophy according to the dimensions of type (functional and structural) and granularity such as unit, integration, and system (3). Dynamic testing involves the creation of test-cases and test-vectors and execution of the software against these tests. This particularly suits the goal of finding design errors, which test cases often matching functional requirements. Comparison of the results to expected or known correct behavior of the software is then performed. Wagner also includes a summary of various statistics compiled on the effectiveness of dynamic testing. His analysis shows that the mean effectiveness of dynamic testing is only about 47%. In other words, over half of potential errors on average are not detected with dynamic testing.

4. Abstract interpretation

Abstract interpretation is the application of mathematical formal methods based techniques to abstract the semantics of a software program. It can be an effective means of performing certain types of software verification. Abstract interpretation as a proof-based verification method can be illustrated by multiplying three large integers in the following problem:

$$-4586 \times 34985 \times 2389 = ?$$

Although computing the answer to the problem by hand can be time-consuming, we can quickly apply the rules of multiplication to determine that the sign of the computation will be negative. Determining the sign

of this computation is an application of abstract interpretation. The technique enables us to know precisely some properties of the final result, such as the sign, without having to multiply the integers fully. We also know from applying the rules of multiplication that the resulting sign will never be positive for this computation.

In a similar manner, abstract interpretation can be applied to the semantics of software to prove certain properties of the software. Abstract interpretation bridges the gap between conventional static analysis techniques and dynamic testing by verifying certain dynamic properties of source code without executing the program itself. Abstract interpretation investigates all possible behaviors of a program – that is, all possible combination of values – in a single pass to determine how the program may exhibit certain classes of run-time failures. The results from abstract interpretation are considered sound because we can mathematically prove that the technique will predict the correct outcome as it relates to the operation under consideration.

5. Code verification

Code verification or code verifiers based on abstract interpretation can be used as a static analysis tool to detect and mathematically prove the absence of certain run-time errors in source code, such as overflow, divide by zero, and out-of-bounds array access as part of the V&V process. The verification is performed without requiring program execution, code instrumentation, or test cases. Cousot describes the application and success of applying abstract interpretation to static program analysis (4). Deutsch describes the application of this technique to a commercially available solution (5). Descriptions of the application of abstract interpretation for software and hardware can be found in both academia and industry. Some of these include:

1. Stack overflow checking of embedded software (6)
2. JULIA for abstract interpretation verification of Java code (7)
3. DAEDALUS for validating critical software (8)
4. Hardware verification of the timing of transistor gates (9)

Color	Explanation
Green	Indicates code is reliable
Red	Run-time error due to faulty code
Gray	Shows dead or unreachable code
Orange	Unproven code

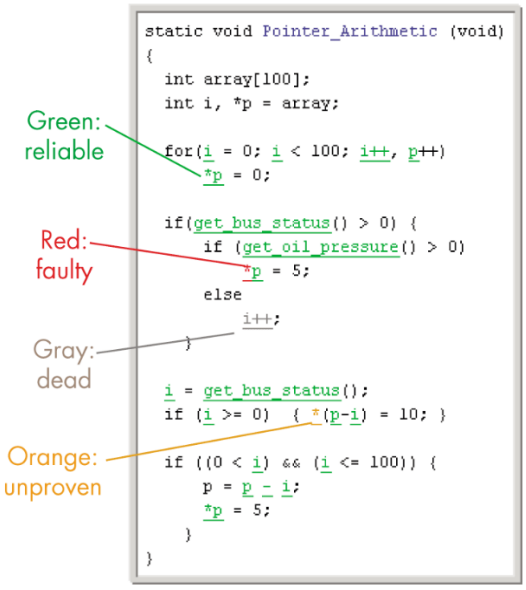


Figure 2 – Polyspace color scheme

To describe the use of abstract interpretation based code verification in this paper, we used Polyspace® (10). Polyspace is a code verifier that detects and proves the absence of certain run-time errors such as

overflows, divide by zero, out of bound array access, etc. The input to Polyspace is C, C++, or Ada source code. Polyspace first examines the source code to determine where potential run-time errors could occur. Then it generates a report that uses color-coding to indicate the status of each element in the code, as shown in Figure 2. Polyspace results that are all tagged in green indicate that the code is free of certain run-time errors. In cases where run-time errors have been detected and color coded in red, gray, or orange, software developers and testers can use information generated from the code verification process to fix identified run-time errors.

Code verification reduces development time by automating the verification process and helping to efficiently review verification results. It can be used at any point in the development process, but using it during early coding phases it enables development and test teams to find errors when it is less costly to fix them. This process takes significantly less time than using manual methods or using tools that require modification of code or run test cases.
Knowing where potential run-time errors could exist or that certain run-time errors are proven not exist helps with productivity. Less time is spent debugging because the exact location of an error in the source code is known. After errors are fixed, verification can be performed on the updated code. Code verification helps developers and testers use their time effectively. Because they know which parts of code are run-time error-free and they can focus on the code that has definite run-time errors or might have run-time errors.

6. Overview of a code verification process

Code verification cannot magically produce quality code at the end of the development process. Code verification is a tool that helps measure the quality of code, identify issues, and ultimately achieve set quality goals. To successfully implement such a solution within a development process the implementers of a high quality software development process must define quality objectives, define a process to match quality objectives, apply the process to assess the quality of code, and then continuously improve this process.

Before one can verify if code meets a prescribed set quality goals, these goals must first be defined. The first step in implementing a code verification process is to define quality objectives. Since the ultimate goal is to create zero defect software, this goal may not be realized in the short term. Therefore intermediate steps and milestones should be established.

The type of code verification that is to be performed is also important. One can perform robustness or contextual verification. Robustness verification helps to show that the software works under all or most conditions or environments and contextual verification shows that the software works under normal conditions or environments. Some verification processes may incorporate both robustness and contextual verification. For example, developers may perform robustness verification on individual files early in the development cycle, while writing the code. Later, the team may perform contextual verification on larger software components. Additionally, quality objectives may involve the use of coding rules or standards and defining software quality levels.

6.1 Robustness verification

Robustness verification proves that the software works under all conditions, including "abnormal" conditions for which it was not designed. This can be thought of as "worst case" verification. Code verifiers such as Polyspace, by default perform robustness verification. In this use-case the code verifier:

- Assumes function inputs are full range
- Initializes global variables to full range
- Automatically stubs missing functions

While this approach ensures that the software works under all conditions, it can lead to unproven results (i.e. *orange checks* in Polyspace vernacular). The software developer or tester then inspects code fragments identified as unproven in accordance with their software quality objectives.

6.2 Contextual verification

Contextual verification shows that the software works under predefined working conditions. This limits the scope of the verification to specific variable ranges, and verifies the code within these ranges. When performing contextual verification with code verifiers such as Polyspace, it will reduce the number of unproven results. Various techniques are available to help with this task.

- Using data range specifications to specify the ranges of function input and global variables, thereby limiting the verification to these cases.
- Create a detailed main program to model the call sequence, instead of using automatically created simple main generators.
- Provide manual stubs that emulate the behavior of missing functions, instead of using the default automatic stubs.

6.3 Choosing coding rules

Coding rules are one of the most efficient means to improve both the quality of your code, and the quality of your verification results. Popular coding rules to choose from include MISRA-C, MISRA-C++, and JSF++ (11). In addition to improved quality, in general the software will be easier to read, maintain and port. Some certification standards also stipulate the use of coding standards and rules (such as DO-178B in the aviation industry). A side benefit is that when enforcing coding standards, improved code verification results may also be observed. In general, use of coding standards and rules does not mean that all rules specified in a certain standard must be enforced.

6.4 Defining software quality levels

Defining and identifying software quality levels is an excellent method by which to create intermediate milestones to achieve a goal of getting to zero defect software. Table 1 identifies an example set of quality levels that can be used as part of a code verification process. The table identifies four quality levels from level QL-1 (lowest) to level QL-4 (highest). Each quality level consists of a set of software quality criteria that represent a certain quality threshold. Quality levels such as these are beginning to be used in the automotive industry. For example, by original equipment manufacturers to stipulate specific Software Quality Objectives (SQO) to their suppliers (12).

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
Document static information	X	X	X	X
Enforce coding rules with direct impact on code verification results	X	X	X	X
Review all systematic run-time errors (e.g. red checks)	X	X	X	X
Review all dead code (e.g. gray checks)	X	X	X	X
Review first criteria level of unproven code (e.g. orange checks)		X	X	X
Review second criteria level of unproven code (e.g. orange checks)			X	X

Criteria	Software Quality Levels			
	QL1	QL2	QL3	QL4
Enforce coding rules with indirect impact on code verification results			X	X
Perform dataflow analysis			X	X
Review third criteria level of unproven code (e.g. orange checks)				X

Table 1 – Example definitions of software quality levels

Static information – Includes information about the application architecture, the structure of each module, and all files. This information must be documented to ensure that the application is fully verified.

Coding rules – This includes the application of selected or all coding rules from standards such as MISRA-C, MISRA-C++, or JSF++. There are two classifications of the application of coding rules. The first set has direct impact on the results of code verification and a second set with indirect impact. Examples of MISRA coding rules that have a direct impact on code verification results are (list is not exhaustive):

- The *static* storage class specifier shall be used in definitions and declarations of objects and functions that have internal linkage
- Conversion shall not be performed between a pointer to an object and any type other than an integral type, another pointer to a object type or a pointer to void
- Floating-point expressions shall not be tested for equality or inequality
- The *goto* statement shall not be used
- Etc.

Good design practices generally lead to less code complexity, which can indirectly improve the results from code verification. The following set of coding rules help address design issues that have indirect consequence on code verification results (list is not exhaustive).

- Identifiers (internal and external) shall not rely on the significance of more than 31 characters
- *typedefs* that indicate size and signedness should be used in place of the basic types
- Braces shall be used to indicate and match the structure in the nonzero initialization of arrays and structures
- Bitwise operations shall not be performed on signed integer types
- Etc.

Systematic run-time errors – These are definite run-time errors that have been identified by the code verifier. They represent errors that occur every time the code is executed.

Dead code – Represents unreachable code. This often highlights a design error, as it questions the presence of a statement which is not reachable under any data conditions.

Unproven code – Indicate unproven code, meaning a run-time error may occur. The three criteria called out in Table 1 indicate threshold levels that should be established by the software development team for code that is not proven to be free of run-time errors. For example, the software team could decide that for divide by zero run-time errors, for the first criterion level 60% of divide by zero errors shall be proven, for the second criterion level 75%, and for the third criterion level 95%.

Dataflow analysis – Identifies errors such as non-initialized variables and variables that are written but never read. This can include inspection of application call tree, read/write accesses to global variables, and shared variables and their associated concurrent access protection.

6.5 Achieving certification

High quality software often implies achieving certification. The verification activities have to be consistent with requirements defined by the certification standards, such as DO-178B or IEC 61508. These standards often mention verification activities such as checking coding rules, finding run-time errors and design errors as relevant steps in the process. Improving the quality up to the proof of absence of certain class of errors naturally fits into these standards. This can enable software development teams reduce alternative verification activities thanks to evidences of these proofs.

7. Deploying an improvement process

In summary, the overall improvement process that can enable software development teams to develop high quality code would consist of the use of code verification as part of the overall software verification and validation (V&V) process, use of coding rules and standards, and identifying milestones for achieving certain software quality levels. For the code verification process, you can also use robustness verification, contextual verification or a combination of the two. It is also important to use code verification early in the development cycle because it improves both quality and productivity. That is, it allows users to find and manage run-time defects soon after the code is written. This saves time because each user is familiar with their code, and can quickly determine why code can or cannot be proven to be reliable from a run-time perspective. In addition, defects are cheaper to fix at this stage, since they can be addressed before the code is integrated into a larger system.

8. Creating high quality embedded software

Critical software defects in the form of run-time errors may be introduced in the coding phase of software development. These defects may not be detected with traditional testing methods applied during unit and regression phases of software verification and validation (V&V). Abstract interpretation based code verification techniques introduce precision and guidance to the software development process. The technique provide precision in guiding software engineering teams to know which parts of code may not fail and isolate those aspects of code that may fail or are most likely to fail. Using these techniques, software developers and testers can minimize effort related to hunting for defects in wide swaths of code. The application of code verification as part of the overall V&V processes is an incremental quality improvement process. This process helps software development teams get on the path of creating zero defect software. It is a robust verification process that helps achieve high reliability in embedded devices.

References

1. **NIST.** *Reference Information for the Software Verification and Validation Process.* 1996.

2. **Bessey, Al.** *A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in the Real World.* ACM, 2010.

3. **Wagner, Stefan** . *A Literature Survey of the Software Quality Economics of Defect Detection Techniques..* ACM, 2006.

4. **Cousot, Patrick.** *Abstract Interpretation: Theory and Practice.* Springer, 1996.

5. **Deutsch, Alain.** *Static Verification of Dynamic Properties.* SIGDA, 1996.

6. **Regehr, John, Alastair Reid and Kirk Webb.** *Eliminating Stack Overflow by Abstract Interpretation.* EMSOFT, 2003.

7. **Spoto, Fausto.** *JULIA: A Generic Static Analyzer for the Java Bytecode.* 1982.

8. **Programme, European IST.** DAEDALUS. [Online] 2002. http://www.di.ens.fr/~cousot/projects/DAEDALUS/.

9. **Viladrosa, Robert.** *Abstract Interpretation Techniques for the Verficiation of Timed Systems.* Thesis, 2005.

10. **MathWorks.** Polyspace Code Verification Products. [Online] 2010. http://www.mathworks.com/polyspace.

11. **MISRA.** The Motor Industry Software Reliability Association. [Online] 2010. http://www.misra.org.uk/.

12. **MathWorks.** Using Polyspace to implement the “Software Quality Objective for Source Code Quality” Standard. [Online] 2010. http://www.mathworks.com/matlabcentral/fileexchange/27525.