

2009

PACIFIC NW SOFTWARE
QUALITY
CONFERENCE



MOVING
QUALITY
FORWARD

OCTOBER 27-28, 2009

CONFERENCE
PROCEEDINGS

TWENTY-SEVENTH ANNUAL
PACIFIC NORTHWEST
SOFTWARE QUALITY
CONFERENCE

October 27-28, 2009

Oregon Convention Center
Portland, Oregon

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

TABLE OF CONTENTS

Welcome	v
Board Members, Officers & Committee Chairs	vii
Planning Committee	viii
Keynote Address — October 27	
<i>Are Lifecycles Still Relevant?</i>	1
Erik Simmons	
Keynote Address — October 28	
<i>Agile, Testing, and Quality: Looking Back, Moving Forward</i>	3
Elisabeth Hendrickson	
Testing Track — October 27	
<i>Reducing Test Case Bloat.....</i>	5
Lanette Creamer	
<i>The Effect of Highly-Experienced Testers on Product Quality.....</i>	21
Alan Page	
<i>Building Alliances.....</i>	31
Karen Johnson	
<i>Testing IPv6 Enabled Applications</i>	39
Travis Luke	
<i>Test Faster.....</i>	47
John Ruberto	
Automation Track — October 27	
<i>Build Robust Test Automation Solutions for Web Applications</i>	61
Wei Liu, Dawn Wilkins	
<i>Too Much Automation or Not Enough? When to Automate Testing</i>	67
Keith Stobie	
<i>Some Observations on the Transition to Automated Testing</i>	79
Robert Zakes	
<i>Web Security Testing with Ruby and Watir</i>	91
James O. Knowlton	

Management, Retrospectives & Requirements Tracks — October 27

<i>Learning from Pragmatic Project Managers: Make Your Project Successful</i>	99
Johanna Rothman	
<i>Can't Travel? Virtual Retrospectives Can Be Effective!</i>	109
Debra Lavell	
<i>Retrospective Analysis and Prioritization Areas for Beta Release Planning Improvement</i>	119
Ajay Jain	
<i>A Distributed Requirements Collaboration Process</i>	129
Brandon Rydell, Sean Eby, Carl Seaton	
<i>Developing Requirements for Legacy Systems: A Case Study</i>	143
Bill Baker, Todd Gentry	

Soft Skills, Process Improvement & Testing Tracks — October 27

<i>Improve Quality by Making Clear Requests and Commitments and Avoiding the "I'll Try" Trap</i>	155
Pam Rechel	
<i>The Elephant in the Room: Using Brain Science to Enhance Working Relationships</i>	163
Sharon Buckmaster, Diana Larsen	
<i>Moving Software Quality Upstream: The Positive Impact of Lightweight Peer Code Review</i>	171
Julian Ratcliffe	
<i>Quality Cost Management — Manage Your Quality Costs or Let Them Manage You</i>	181
Ian Savage	
<i>The Search for Software Robustness</i>	195
Dawn Haynes	

Testing Track — October 28

<i>Half-Baked Ideas for Rapid Test Management</i>	205
Jon Bach	
<i>Best Practices for Security Testing: Top 10 Recommended Practices</i>	213
Aarti Agarwal	
<i>Why Tests Don't Pass</i>	229
Douglas Hoffman	
<i>Visualizing Software Quality</i>	237
Marlena Compton	
<i>Everyone Can Test Performance</i>	247
Scott Barber	

Quality Techniques & Data Mining Tracks — October 28

<i>Software Pedigree Analysis: Trust But Verify</i>	249
Susan Courtney, Barbara Frederiksen-Cross, Marc Visnick	
<i>Reconfiguring the Box: Thirteen Key Practices for Successful Change Management</i>	263
Leesa Hicks	
<i>Leveraging Code Coverage Data to Improve Test Suite Efficiency and Effectiveness</i>	275
Jean Hartmann	
<i>Where are You in Usability?.....</i>	289
Kelcie Anderson	
<i>An Empirical Study of Data Mining Code Defect Patterns in Large Software Repositories</i>	295
Kingsum Chow, Xuezhong Xing, Zhongming Wu, Zhidong Yu	
<i>Data Mining for Process Improvement</i>	307
Paul Below	

Agile & Process Improvement Tracks — October 28

<i>Distributed Team Collaboration.....</i>	317
Kathy Milhauser	
<i>Moving to an Agile Testing Environment: What Went Right, What Went Wrong</i>	327
Ray Arell	
<i>Managing Software Debt: Continued Delivery of High Value as Systems Age</i>	335
Chris Sterling	
<i>My Experience at Adopting an Agile Software Development Approach</i>	341
John Bartholomew	
<i>Improving Your Quality Process: A Practical Example</i>	349
John Balza	
<i>I Have Two Managers?!: One Company's Model for a Consultative Testing Team and Matrix Management</i>	361
Amy Yosowitz	

Software Quality Track — October 28

<i>The Myths of Rigor.....</i>	371
James Bach	
<i>Score One for Quality: Using Games to Improve Product Quality.....</i>	381
Joshua Williams, Ross Smith, Dan Bean, Robin Moeur	
<i>New Challenges to Quality in the 24 x 7 Enterprise IT Shop: Post-Integrated Business Automation Systems</i>	395
Al Hooton	
<i>Holding Our Feet to the Fire.....</i>	405
Jim Brosseau	
<i>ProdTest: A Production Test Framework for SAAS Deployment at Salesforce.com</i>	415
Bhavana Rehani, Kei Tang	

PACIFIC NW SOFTWARE QUALITY CONFERENCE

Earlier this year I was waiting outside a local Portland restaurant. While I was waiting a vintage Ford Mustang came rolling into the parking lot. That doesn't sound too exciting, however, the car pulled forward and backed into a parking spot right across from the front door of the establishment. Again, not too exciting. Cars come in, park, and leave all day long. What caught my eye, and made me smile was the front vanity license plate; it read QUALITY.

So I naturally made the connection between our conference theme "Moving Quality Forward" and the symbolism of the car.

Welcome to the 27th annual Pacific Northwest Software Quality Conference. This year we have moved our conference to a more intimate setting at the World Trade Center which we believe will better support our mission to enable knowledge exchange to produce higher quality software.

How are we planning to move quality forward? Kicking off our conference this year is a very talented and experienced speaker, Erik Simmons from Intel Corporation. He will be sharing his perspective with his Keynote: "Are Lifecycles Still Relevant?" His work at Intel over the past 10 years uniquely qualifies Erik to challenge traditional thinking of how and why lifecycles are a part of product development. Our second Keynote speaker is Elisabeth Hendrickson from Quality Tree Software. Elisabeth has been in the SW industry since 1984. She will be focusing on sharing her experiences in agile, testing and quality. Elisabeth will look both forward and backward to highlight practices that have gained adoption, as well as share emerging trends to see where our profession is going next.

Our conference is full of practical, useful and valuable information. With everyone's help, SW quality is beyond doubt moving forward. Glad you are here in the driver's seat with us.

Debra Lavell

2009 President, PNSQC

BOARD MEMBERS, OFFICERS & COMMITTEE CHAIRS

Debra Lavel — Board Member & President

Intel Corporation

Bill Gilmore — Board Member & Vice President

Robert Cohn — Board Member & Secretary

Wacom

Doug Reynolds — Board Member & Treasurer

Tektronix, Inc.

Chris Blain — Board Member & Marketing Communications

Tranxition

Marilyn Pelerine — Board Member & Community Outreach

Symetra Financial

Ian Savage — Board Member & Audit

McAfee, Inc.

Paul Dittman — Program Chair

McAfee, Inc.

Kathy Iberle — Program Chair

Hewlett Packard

Doug Hoffman — Invited Speaker Chair

Software Quality Methods

Ellen Ghiselli — Volunteer Chair

McAfee, Inc.

Shauna Gonzales — Luncheon Program Chair

Nike, Inc.

Rhea Stadick — Operations & Infrastructure Chair

Intel Corporation

Patt Thomasson — Communications Chair

McAfee, Inc.

PLANNING COMMITTEE

Darrell Bonner

John Burley

Ken Doran

Moss Drake

Daisy Dental

Joshua Eisenberg

McAfee, Inc.

Cynthia Gens

Frank Goovaerts

Tektronix, Inc.

Sara Gorsuch

Symetra Financial

Les Grove

Tektronix, Inc.

Brian Hansen

Radar Engineers, Inc.

Miska Hiltunen

Randy King

Diana Larsen

Futureworks Consulting

Dave Liesse

Launi Mead

Symetra Financial

Jonathan Morris

Kronos, Inc.

Carol Oliver

Bill Opsal

Dave Patterson

Ganesh Prabhala

Intel Corporation

Wolfgang Strigel

Stephen Udycz

Are Lifecycles Still Relevant?

Erik Simmons
Intel Corporation
JF5-113
2111 NE 25th Ave.
Hillsboro, OR 97214-5961
erik.simmons@intel.com

Author Biography

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

Abstract

Software lifecycles have been a topic of interest within software development for decades. Many improvements in software development practices have been dependent on, or even based on, new or improved software development lifecycles. More recently, the focus has shifted to practices, reducing the attention paid to lifecycles. For example, Extreme Programming has a set of practices as its central focus (pair programming, test-first design, continuous integration, etc.).

In the agile and post-agile software development world, are lifecycles still relevant? If so, in what ways, and how can lifecycles be modified so as to remain valuable as software development continues to evolve?

One barrier to answering these questions is that there is no consensus definition of *software lifecycle*. Common definitions within available books, papers, and websites include lifecycles as a process, a set of practices, a method (or methodology), a period of time, and a series of phases and gates. This creates a tangle of terminology and some problematic overloading of the term "lifecycle".

Lifecycles can also be viewed in several other ways not found in the common definitions, including: a series of commitments or decisions; a series of models that culminates in the final system; a description of the flow of information among groups, specialties, and systems; a set of value-added interactions among producers and consumers (marketers, designers, coders, testers, etc.); a set of states through which the project or product must pass; and a "map" of the project's journey from start to finish, including milestones and points of interest along the way.

The question that comes to mind is whether any guidelines exist for what a lifecycle ought to contain – especially in relation to the current and future challenges of software development.

Software is being called on to solve increasingly complex (as opposed to merely complicated) problems. The result is more complex software and more complex software development projects. The software lifecycle needs to reflect this challenge. Complexity in general, and Complex Adaptive Systems in particular, will exert a strong influence on what happens next in software development. Already, interest in “edge of chaos” development techniques is growing. Proponents advocate that “edge of chaos” development increases innovation, creativity, and even productivity.

Improvement may come from thinking of and treating the lifecycle not only as a small part of the larger system that creates software, but as a system unto itself as well. A lifecycle is not just “the diagram”, or “the process” as part of that larger system. The lifecycle includes a social dimension in addition to work products, processes, state definitions, and milestones (as shown by the inclusion of decisions, commitments, interactions, and value in the ways lifecycles can be viewed above). Said another way, a lifecycle contains cultural, structural, and political information; organizational and team structures, decision making policies, values and beliefs all are potentially within the scope of a lifecycle. Further, a lifecycle must evolve to meet a changing mission over time, managing its resources to ensure that it remains effective and viable to the organization. It loses much of its power when it is reduced to a picture or treated as no more than a prescriptive process.

Lifecycles have not lost relevance. However, the nature of software development is changing in ways that make a holistic, systems-oriented view of the lifecycle more powerful, and more important, than ever.

Agile, Testing, and Quality: Looking Back, Moving Forward

Elisabeth Hendrickson **Quality Tree Software**

Once considered radical, Agile approaches have become mainstream, with Scrum as the most well-known and most adopted of the Agile methods.

For the last several years, Elisabeth Hendrickson has worked with a variety of Agile teams, some for whom agility came naturally and others who struggled with the transition. In this keynote, Elisabeth will:

- Discuss common testing and quality related challenges for Agile teams.
- Highlight practices that have gained widespread adoption and practices that turn out to be easier said than done.
- Examine why some teams find it easier than others to make a real transition to Agile.
- Look at how tools have changed to support Agile.
- Explore emerging trends to see where we're headed next.

***Elisabeth Hendrickson** is a consultant specializing in Agile Testing. She started in the software industry in 1984 and has held positions as a Tester, Test Manager, Test Automation Manager, and Quality Engineering Director.*

Elisabeth founded Quality Tree Software, Inc. in 1997 to provide training and consulting in software quality and testing. She's written numerous articles and is a frequently-invited speaker at major conferences.

In 2003 she became involved with the Agile community. These days Elisabeth splits her time between teaching, speaking, writing, and working on Agile teams with test-infected programmers who value her obsession with testing.

Reducing Test Case Bloat

By

Lanette Creamer



Adobe Systems, Inc

2009

Author Bio

Lanette Creamer has been with Adobe Systems since 2000 testing products such as Adobe InDesign versions 1.5 through CS, Adobe InCopy 1.0, Shared Technologies across applications such as XMP, XML, and has been working as the Quality Lead for the Adobe Creative Suites Workflow Team since 2006, promoting a delightful user experience across products. Lanette studied Graphic Design at Western Washington University, but her true love was for Photoshop, Illustrator, and PageMaker. After attending an inspiring seminar at the CAST conference in 2007, she started a testing blog at <http://blog.testyredhead.com/> hoping to find other people who are passionate about testing. Her technical paper, published in the 2008 Pacific Northwest Software Quality Conference Proceedings and corresponding presentation, "Testing for the User Experience", was selected by attendees for a follow-up presentation to close the conference.

Introduction

We may be ready to move on to the future and work on the “new” and “innovative” features that are being created. For many of us, if we don’t deal with the past, it can easily come back to haunt us, slowing down the creation of new projects and robbing our testing time unexpectedly, often to the point that testing becomes the bottleneck that slows innovation to a crawl.

For those of us who work on software that already exists, exciting new functionality and improvements are the main things that drive upgrades, as well as compatibility with new platforms. However, if end users can’t trust the quality of the legacy features they rely on, they will be reluctant to upgrade, or worse, your new versions will get a reputation of being “unstable” harming overall adoption. Embarrassingly, in some cases users will downgrade their software to an earlier version because they are unhappy with the quality of the newer release. Some users will go so far as to ask for a product that is no longer sold trying to get to a version that they feel is reliable, dragging the company reputation through the mud in the process.

This paper is not specifically about the logical part of testing or about proving that some of your tests are unnecessary using formal mathematics or “combinatorics”^{1, 2} to prove some tests are inefficient. It also is not a step by step instruction list you should follow for the most technically efficient list of test cases which you can prove without a doubt are best to cut for your project.

This paper is about the subjective and difficult part of testing which has no provable mathematical “correct” answer. It is about risk management, test planning, cost, value, and being thoughtful about which tests to run in the context of your specific project. I’ll cover why to reduce test case bloat, when it can be done, who does it, along with a few examples that I’ve used in practice. I’ll share the criteria we used to reduce our test cases when faced with the challenge to cover over three times the applications while our test team was reduced from sixteen down to four testers from the previous product cycle.

As testers we are actively participating in building the future of software. We are facing increasingly complex software as we are building new features on top of software that is maturing. While the scope of our testing is expanding, we are often expected to “do more with less”. In these tough economic times we often must balance testing existing features that customers rely on every day with potentially risky new features and interactions. How we address testing legacy code has a significant impact on the quality of future software.

This overview and experience report can help you consider the legacy test cases you are maintaining and running. If you would like to spend more of your time developing new test cases and working on the exciting future of software, consider which test case reduction techniques are the most practical in your context to allow you to test legacy features more efficiently.

1.0 Test Case Bloat

1.1 Why is Test Case Bloat a Problem?

Ideally, you would wrap up a software project, and head right into a new project with your focus firmly on the future. That is much easier to do when you aren't interrupted by the need to test a security patch, a dot release, and a bug fix for a critical account. Even if we did manage to ship perfect software, we still can't predict future environmental changes. A new operating system, browser, or other software our product relies on could change, requiring rework and retesting. While it is impossible to test everything dealing effectively with the past is what can allow us more time to focus on the future. This paper is about one aspect of dealing with legacy software, specifically how to reduce test case bloat.

1.2 What is a Test Case?

Clinically defined a test case is an input and an expected result.³ For my purposes it doesn't matter if a test case is automated or manual so long as it is a planned test. For the purpose of reducing test case bloat, I'd go further and say that it is a test you plan to execute a minimum of once in the product lifecycle.

1.3 What is Bloat?

This is absolutely the million dollar question and quite a tricky one. Test case bloat is any test where the cost of keeping the test is higher than the cost of getting rid of the test for any reason. Even if the reason is simply that the area isn't very visible or used in the software.

Many years ago I may have told you that test case bloat was simply running any test that didn't result in a likely bug. My opinion has changed vastly since those early days of my career. I very much hope to be running critical tests that do not find bugs. As part of the overall software development process, when some sanity tests never fail, that simply means that the stability of existing code has improved upstream. It does not necessarily mean that those cases are "bloat". If failure of the test would be a test blocking bug and catastrophic failure for the software, it is critical to continue running the test and only lower the cost of running it often and earlier.

There are some test cases which result in bugs that I now consider bloat and would cut entirely. In the past I would have kept them simply because I believed that any test case that revealed a bug was vital. However, finding bugs that a user is unlikely to encounter

and is not going to be fixed means effort was expended for no quality improvement. How expensive is it to create, execute, and maintain that test? Is that cost greater than the cost if the test fails and it is only later discovered by users?

There are times when what you cut may not be bloat. There are some situations where the decisions are the equivalent of “Do we cut off the arm or the head?” Well, a person can live without an arm. If you are in a situation where you are so time constrained that critical areas will be untested, you can still communicate the risk, be transparent and use a strategy to test the most important areas first. It is possible to plan for and do testing for a very time constrained project.

On my test team after releasing CS3, the largest product in the history of Adobe at the time, our company had a reorg to align to new initiatives. While we were very lucky to have minimal layoffs, our team went from 16 down to 4 testers as our people were transferred to other projects. Rather than working on the entire Suite, they would be working on Suite components and technologies, leaving just a few of us with a focus on the Suite itself. While this made sense for the changes at the company, it seemed a very daunting task to perform testing with less than 1/3rd the size of the test team remaining to test a collection of products that had grown from 4 to 16 products during the last project. My immediate reaction was despair and a serious concern that we would fail. I did realize with time that while we could not test as much as we tested before with so few people, but we certainly could accomplish important testing. In these situations it helps to take emotion out of the picture, and remember that by making an informed test strategy and executing on it, that no matter what the resource constraint is, you will be directly having a positive impact on product quality. It is even more important to get the priorities right and the bloat out of the planned testing, or critical testing could happen too late, or not at all.

In some software there are vital tests which must always be run so that testers are up to date on the status of these critical features.

Some examples of existing test cases excluded from being possible bloat:

- external regulations an application must adhere to
- most software tests which cover security
- safety
- legal matters
- any test which would be catastrophic and highly visible if it were to fail

Basic build sanity tests are the type of test case that are generally not test case bloat even if they are not finding bugs often. At a base level, to be testable, you need a certain number of test cases to pass. While these aren't the most complicated or esoteric and clever tests you can create, they are critical and must pass for others to rely on a consistent base level of quality to start testing from. Other than the list of exceptions started above, the rest of the existing tests cases, automated, tool assisted, or human executed, are possible candidates to be cut.

By asking the following questions, we were able to change the focus of what we were testing and we ended up not only completing the testing we agreed to execute, but we also managed to run some tests for another team. When are legacy test cases run? How often are they run? What is the cost of maintaining this test? What is the risk of not running this test? Can or should someone else run this?

1.4 When is Bloat reduced?

No matter what software development method is being used, it is possible to reduce test case bloat whenever test planning happens. If that is every week, every month, once per product cycle, or at each product milestone, whenever you are creating test cases, planning for when you are going to test, or preparing a set of tests to run for any milestone, sprint, or certification, it is a good time to think about what percentage of your time will be dedicated to testing legacy code and features, and how you are going to focus on those test cases which will have maximum return. What will you want to consider again for this product, and what are you willing to cut forever and why?

There are many ways to reduce test case bloat, but rather than drawing straws, using the magic 8 ball, or a random number generator to delete test cases via a lottery, I'll cover eight ways that I know of to reduce the total number of tests you are planning to run. With this list to get you started, you can add your own ideas and come up with the best possible combination of preserving the best legacy test cases that still apply.

2.0 Methods for Reducing Test Case Bloat

2.1 Mathematical Combinations

All-Pairs Testing, Pairwise Testing, and orthogonal array, are all variations on the idea that you can use math to find the best combinations of inputs and expected results to create an efficient subset of tests to cover functionality. While many of us use equivalency cases when creating test cases, it can be helpful to use them when reviewing existing test cases or planning to run legacy test cases. Make sure that you are running only the minimum set possible using math by using only the combinations suggested by the tool you run instead of a full set if presented with a full coverage matrix or set of test cases designed to cover all combinations of inputs. This isn't perfect and there is some risk that you could miss.

As a bonus, lots of engineers just love math and will be very pleased that there is a concrete reason for this set of tests based on numbers, so this can be a low pain way to reduce the number of test cases in some contexts. There are some free tools to help, as well as available commercial tools.

In practice this wasn't as useful as I'd hoped in reducing testing scope for my particular project, but because the most math oriented people love it, it was worthwhile to use where possible. The reason math based equivalency may not result in significant bloat

reduction is that equivalency pairs generally apply only to functional test coverage, which is a small percentage of the total tests which are run on most software. It is my experience that you can eliminate some legacy test cases safely this way depending on the design of your legacy test cases.⁴

I think it is a useful way to reduce test case bloat in the instances where someone expects full test coverage of all inputs in some UI. Only a small percentage of the kind of legacy testing that teams at Adobe are tasked with fit this description. Even if it was an entirely perfect best practice, it would not address a majority of the testing challenges we face.

Math alone doesn't solve much of my testing problem, so getting a free tool to do reduction for me is how I include this method in the small percentage of cases it applies in my current work. I wouldn't claim to have a full understanding of it, nor would I be interested in a debate for or against it as a useful practice. For those of you who are interested in the arguments against, check out the article, [Pairwise Testing: A Best Practice that Isn't](#), written by James Bach from PNSQC 2004 which points out some of the flaws in blindly applying tools and math without considering the context.⁵

If you are looking for mathematical programs to help you reduce tests based on math, please see the excellent collection of easy to understand combinatorial testing papers at <http://www.combinatorialtesting.com/clear-introductions-1>.⁶

2.2 Tiered by Importance

One method of dealing with bloat is by organizing the test cases you have into more manageable sized pieces so that time can be allocated in a more strategic manner. While organizing by importance to the project is pretty common, usually it is done as a way to organize test cases rather than to reduce bloat. The idea is that when your test cases are created, you add an extra step where you create a tiered system that works with your method of test case storage. I've seen systems based on when a feature is expected to work and the tiers marked as Alpha, Beta, FC, Milestone 2, Stage 4, M4, July, and Sprint 4. It doesn't really matter how long the iteration or what the marker is called, if it is based on time it is a generally a tiered system for availability, but not always tiered for importance.

While it is vital to know when something is expected to be testable, for test cases I'd suggest an importance tier as well which is pretty standard. So long as the categories reflect how important knowing the result of that test is to the project, it is tiered by importance. The advantage of knowing the importance of your test cases is that you have a built in contingency plan if you must cut tests due to time or resource constraints. You can get agreement and be transparent about your test plans and you can be more confident that what you see as critical to the quality of the project is consistent.

Importance is going to mean different things depending on the surrounding factors. In many cases importance will vary depending on which stakeholders are most critical to the success of the project.

2.3 Stakeholders

- End users-When the results of the test case would impact a large number of users. For some software you can evaluate usage data for legacy tests to more accurately prioritize the impact the test would have for users.
- Internal Customers-When teams within a company rely on the results of this test case. This is very common when components are integrated or shared. If a test case is covering the ability to integrate the importance raises with the number of expected integrations.
- Test-When a failure would block other testing.
- Shareholders-The result of this test represents a major loss of revenue to the company. Many security tests may have more importance for this reason. For some software this means that ad revenue or license revenue depends on the result of this test case.
- Marketing-Product demos could fail or be delayed if this test case fails. A public demo failure is likely to have a different priority than the same result in a test lab.
- Certifications-When any external certification (Logo, operating system, device) could be denied based on the results of the test case.
- Legal-There could be a lawsuit or any undesirable legal result associated with this test case.
- Other-In your specific situation there will quite likely be stakeholders not listed here that need to be considered.

As part of your plan to reduce bloat, it can be helpful to state your assumptions about who is important and where you are placing testing priority and why. When reducing test case bloat you are taking a calculated risk. You are weighing the risk of being unable to test new features by insisting on testing every legacy case against the risk of purposefully not running some tests. When you share your starting assumptions with your stakeholders you offer them the chance to counter with their own assumptions and often you can clarify the boundaries of your testing this way to avoid gaps in testing or duplication.

2.3.1 Test Case Reserve

Some tests are so fundamental to the success of a project that they must be run before any release to customers and in some cases on every build. When reducing test case bloat it can be helpful to know which tests are entirely off limits. I've seen different approaches to this case. On one project there was an automated Build Verification Test called "minimal" and each developer had to run the test suite before they could check in their code. However, in order to improve product quality, "minimal" kept growing. Each

release, more and more features were added to “minimal” and it morphed into what should have been called “maximal”. Not only did this bloat the builds, but it slowed development. Had the bar been higher for the tests allowed in to “minimal”, not only would the developers feel it was more helpful, but the time spent on adding more and more tests could have instead been used to make the validation more effective, the tests faster, and the results more accurate and easy to isolate. This is the ultimate goal of reducing test case bloat, to test with more focus, and rather than testing more, to test what is more important.

When organizing test cases it can be helpful to track which cases fall into this category and for what reason. Then in your test planning you can focus your strategy on moving these tests upstream in the development process, using these tests to measure development progress, and working towards faster notification and resolution of breakages in these critical areas. Reducing the number of tests that fall into this category can allow you to spend more time improving the tests which really add the most value to the project.

Does this mean that some important tests do not get run that should be? Yes. It is well known that it is impossible to test everything. Is it irresponsible to know of a test case and decide not to run it? We know of test cases and decide not to run them every time we design test cases, when we define the scope of our testing, use math to reduce cases, or remove support for a configuration. I challenge you to think about how wrong it is to steal time from the great new test cases that are more important because you are unwilling to let go of test cases which have lost their relevance or are no longer adding significant quality improvement or customer value. It is easy to think about the risk of what we do, but it is difficult to consider the risk of what we do not do. Running the same set of tests because they “must be run” without considering why is risky. I am not suggesting that you throw out all of your legacy test cases, only that you consider the risk and decide what works best for your test project.

If you take the list above under Stakeholders and find a good reason why that test case must be run often (for example, on every build or before every release to a customer), add it to the test case reserve, or collection of tests which are not to be considered for cutting unless the reasons they are critical should change. When collaborating with other testers or handing off a test area to other personnel, it is helpful to understand why a percentage of tests are considered critical to the project.

2.4. Change Based

It’s widely accepted that with each code change there is a risk of introducing bugs. For this reason there are a number of ways to use code changes to predict “hot spots” or areas of the software under test which are high risk, especially when testing legacy software. Using information from a change list is a basic example of change based testing that most of us use regularly to guide our testing. In the case of a dot release you can look at the bugs fixed and have some guidance as to which areas are most likely to be impacted.

If you have your test cases categorized by feature area, you can focus the majority of your testing on the areas of change first, only reserving a small percentage of last minute tests for areas of lower risk and less change. This reduces bloat by addressing the “test and retest” cycle that can happen on software projects when every test is considered critical to run at all points during the project.

One other technique which is based on change is to plan major testing events around points of major change. One example of timing testing around changes is that after major integrations we will run an automated suite of sanity tests, and if those pass we will then run broad collaborative user workflow test exercises to get broad coverage and this gives us information on the user experience across multiple products complete with stability information.

While there are many barriers to implementing a visual map showing all changes across multiple products and platforms, I've often dreamt of a useful map of all changes per week across products, showing all component integration, bug fixes, and feature check-ins for all products. Even having this information for one product would be helpful to a tester. I have not seen a code check-in tool create such a visual map that is human readable, but I can imagine that one day such a feature or tool might exist and guide test professionals to more efficient focus on the areas of highest risk. I see this as the “laser guided missile” and right now, at least on my team, we don't have that clear a picture of the code changes going in to the software under test. We can use the data we do have to send out unmanned drones.

We can also use all of the change data we are able to compile to show us areas of low change. These areas are more likely to be stable than areas undergoing major change, so we can look for test cases in these lower risk areas as candidates to test less often, or even not at all if that amount of risk is acceptable for the project.

2.5. Timing Based

While it generally doesn't sound so great to test managers, strategic procrastination really can be an excellent plan. If you know for a fact that an upcoming change will invalidate the results of certain test cases, it may be better to hold off testing them until after that change has occurred. When you are waiting for a code change, a feature to be testable, or another test to be run, it can be helpful to label the set of tests clearly with what you are waiting for such as a milestone, a feature hand-off, or a calendar date.

I recommend labeling a set of test cases using the exact feature or milestone I'm waiting for rather than the expected date to start running the test cases because it is often easier to maintain. The test triggering reason remains relevant even if the schedule, ownership of the feature, or even the owner of the test case changes.

There are some situations where it is the relevance of a test case is time sensitive and a test case should expire. If your test cases are ranked by priority and you have reached

point in the project where failure of a test case will not possibly result in a bug fix, the test case has far more value if results are reported earlier. It is best to note the expiration date, run the tests when the results can have a positive impact on the software quality, stop testing, and retire those test cases for the rest of the cycle.

2.6. Test Case Find Results

In any set of tests, we usually have test cases which result often in bugs and then we have those duds which have never found a bug of importance. With some exceptions, you may want to consider a results based restructuring of your test cases. Top performing test cases get run first, and possibly more often. Those lowest yielding tests should possibly be put on the bench of “backup” tests. If these tests continue to take time and energy, but do not generate bugs or protect a key area which would be a ship stopping bug if the test ever failed, put them into consideration for retirement based on their contribution to overall quality. In our test case database, we have an “inactive” status which means a test case is not being run currently, but is not deleted and can still be viewed. The past data and the test case itself is tracked, including who moved it to the status of inactive and when, but it is no longer taking up tester time and money to be maintained.

Be careful when considering test cases that you don’t accidentally throw out a test case which exists only because failure would have a huge customer impact. What we really are evaluating the test cases for is “Return on Investment”, and there are some cases which would be so expensive if they ever failed that even if they haven’t ever failed in the past, the risk that they could is too great to take. Past results is only one criterion to use when deciding which legacy test cases receive focus. The impact of failure of that test case happening post ship without your test team knowing the results in advance should be a key consideration.

2.7. Selective Combinations

How many code paths does this test case cover? When designing automated test cases, it is often most efficient for isolation purposes and flexibility to run tests in parallel to methodically test one thing at a time. This can make maintaining the tests and fixing the bugs simpler, but it often will miss bugs that happen when a combination of factors are in play or a series of steps causes a bug rather than one action in isolation. In addition to the properly isolated tests which are designed to cover functionality, I like to compliment that set with a complex scenario which provides broad coverage. The advantage to having a few complex scenario tests which represent a large more detailed set of test cases is that the test can be run by a human to give thoughtful evaluation. Once the more isolated and scripted tests can be successfully automated, having the test suite pass can give you confidence that nothing has broken behind you. However, even very well designed automated validation will have gaps in coverage, so having a few broad tests which combine features in a complex and user focused way can efficiently add to your confidence without duplicating tests already covered by automation.

For example, at one point I was testing the integration of XML parsing core technology to ensure proper import and export of UTF-16 encoded characters. In order to cover the basic functionality, I wanted to ensure we supported every character that the spec supported.

2.7.1 Example

Automated Test Pass-New

Import XML files into InDesign for each character supported in UTF-8 and UTF-16 both in content and tag name.

Export the file and Diff.

Expected: The XML file should be the same.

In this example, each character is processed in a separate file because this makes it very easy to isolate any parsing problem instantly, and also makes the automation more reliable, as if one file fails, the rest of the tests still will continue running.

Automated Test Pass-Legacy

Once all of these files passed, it was very unlikely they would break behind us, yet with each integration I continued to run these tests. Why not? They still counted as “test cases run” and that should give us some confidence.

Instead of running each file through one at a time, I combined all of the files into one XML file for verification of encoding. What this did was gave me just one “Pass or Fail” result summary. If it ever were to fail, I could run the more isolated cases to easily find out why, but I no longer got a pass result for each character. Instead, I asked the question after integration, “Does parsing and exporting UTF-16 encoded characters still work?” For the rest of that release the answer was either “Yes”, or “Total failure of all characters either with importing or exporting.” When you are maintaining legacy code, sometimes a summary is better than a detailed question. In addition to combining the encoding test, I was able to get some wonderful customer content, to test for parsing, test for performance, transformations on import, results of merging and appending with existing content, and the application of style mapping by making one summary test to run to look at the legacy functionality. The advantage of testing a legacy feature is that you have customer feedback, usability data, and often times real world files to test with. Combining the functional tests of the past with some user content can create a useful summary scenario.

You may not need to do a thing with your tests themselves in order to summarize your results. Some test case management systems allow you to customize collections of tests and get only a pass/fail for a specified group. If you are working with such a system, you may just need to organize your tests in a way that makes sense for the

maturity of the code you are working on so that you are notified only when there is a change. People don't use software one feature at a time in an isolated and methodical way, so combining tests often more closely matches the user experience. When reducing test case bloat, look for duplication. Are any of your simple tests already part of your build validation?

Human evaluated summary test scenarios can be to stack one test on the previous test, emulating a more natural user workflow, and that will make running the tests faster, more efficient, and also much more realistic for evaluating the user experience.

We use a custom test case management system at Adobe, which means we can ask for custom fields for tracking test cases for each product. It is possible to create a broad summary case which contains smaller and more isolated test cases within, yet report results against the entire set of cases per test run. If subsets and grouping aren't possible with the tracking tool you are using, you may be able to use a simple keyword per scenario to mark which isolated test cases are covered in the summary scenario. Despite the administrative overhead you may have to endure if your test case management methods do not include bulk updating, you still may save time over running each of the tests in isolation by creating a realistic summary scenario.

2.8. Model Based

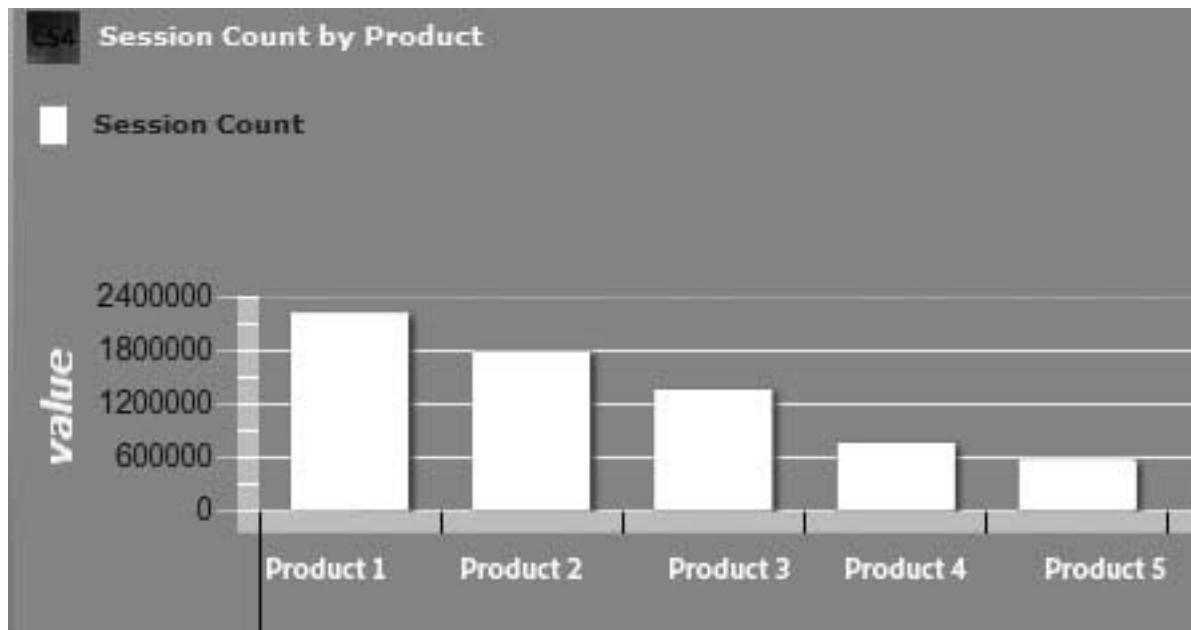
Models are being used in software design, automation design, test planning and even results reporting. There are some widely model types used, such as UML models of machine states and user scenario diagrams, as well as some less known models, such as large flow-charts which combine state and user data, and even some artistic cartoon diagrams with stick figures. Since attending a presentation and follow up discussion on Model Based Testing⁷, by Harry Robinson at StarWest 2006, I've used models to help with some aspects of my overall test effort. While I have not yet created my own robot army based on Harry Robinson's ideas, I did use some models to help reduce the number of legacy test cases we were covering for XML export in 2006, and again in 2008. For more reading on Model Based Testing, see http://www.geocities.com/model_based_testing/online_papers.htm

When reducing test case bloat, consider using the models that you have to evaluate your test coverage and examine where you could possibly take more risk, avoid duplication, leverage existing testing based on what is close, or what you could easily "summarize" as explained in section 2.7 based on stability of the area. If you do not currently have any models, check out the above list of online papers and consider which type of models might be helpful in your situation to visualize your overall testing in order to plan where you are going to focus and also where to cut bloat. My experience with models currently includes mainly mapping user workflows⁸ through multiple applications which is much more like making a map from Point A to Point B with scenic viewpoints along the way than it is creating an entire topographical map of a previously unexplored area. Regardless of the type of model, when you have a visual summary of a feature, you can identify areas of overlap, areas of risk, and possibly areas of interaction where you might want to look for duplicative test cases.

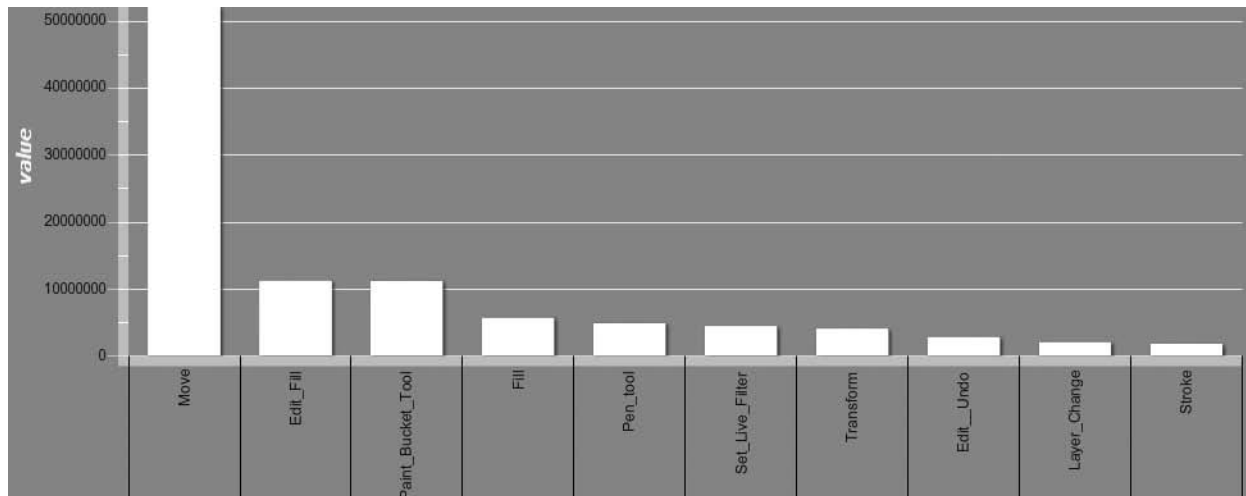
2.9. Customer Workflow Based

The most effective way that currently reduces our legacy test cases is based on customer impact. We combine usage data coming in from our customers allowing us insight into which products they use most commonly, how long each session lasted on average, as well as which were the top ten features used in each product. Future features must also be considered as they significantly impact legacy features as existing functionality changes, and new abilities create previously impossible workflows.

Having objective data confirm what we could only assume previously gave us the courage to reduce our test coverage in some legacy areas so that we could focus our energy on the high impact features, including the new features, areas undergoing major change, and very visible features that impact a large percentage of users. In addition to reducing test case bloat, for the features we discovered were more widely used it was easier to justify the time to automate that test and add it as a build verification test. With the top 5 features covered to some level before the build even passes acceptance, we can focus on the way that the important legacy features interact with the new functionality being added which gives us more time to evaluate the user experience further upstream. The sooner we can offer feedback on new features as well as the impact they have when they are inconsistent with existing workflows, the more cost effective it is to make changes.



Product Usage



Top 10 Features Used in Product

Summary

Any test case that you decide not to run, or to run less often introduces some level of risk that you will miss a bug. However, each legacy test case comes with a cost to run which introduces expense to the project. Evaluating your legacy test cases to reduce redundancy, prioritize, and summarize can allow you to move forward to testing new functionality.

Prioritizing tests can help save time by running the same legacy tests, just less often. If the test doesn't fall into one of the most used workflows or one of the top priority experiences that the software is delivering, it isn't in the top tier. A test being in the top tier means that the test failure will stop shipment of the product. The next tier of tests should be run until Release Candidate submits. Bugs in this category most likely will be fixed if the test fails so long as it can be fixed without jeopardizing the product schedule. The final tier is any test case which must be run during the product cycle which doesn't fit into the first two tiers.

Set out in your test plan what legacy workflows you are protecting and what you are consciously deciding not to test, and why that decision was made. When you have the guts to not test the low priority areas and get stakeholders to sign off on it, you are recognizing and avoiding a larger risk. Adding more and more test cases without ever taking any away is an even bigger risk. If every legacy test is of equal importance, with every new version you need either an exponentially growing QE staff, a schedule that gets longer for testing each time, or you reduce risk by taking on less ambitious new functionality.

References

1. Bellman, R. and Hall, M. Combinatorial Analysis. Amer. Math. Soc., 1979., <http://mathworld.wolfram.com/Combinatorics.html>
2. Combinatorialtesting.com overview, Bolton, Michael Pairwise Testing, DevelopSense©2004
- Videos at <http://www.combinatorialtesting.com/videos>
3. IEEE (1998). IEEE standard for software test documentation. New York: IEEE. ISBN 0-7381-1443-X.
4. P. Black, "My Life with Bugs, or Why I Believe in Combinatorial Testing", American Society for Quality, Gaithersburg, MD Oct. 30, 2007.
5. James Bach, Pairwise Testing: A Best Practice that Isn't, PNSQC, <http://www.testingeducation.org/>, 2004.
6. Robinson, Harry, http://www.geocities.com/model_based_testing/
7. http://en.wikipedia.org/wiki/Model-based_testing
8. Creamer, Lanette, Testing for the User Experience, PNSQC, 2008.

The Effect of Highly-Experienced Testers on Product Quality

Alan Page
Director, Test Excellence
Microsoft

Abstract

In order to keep up with improvements in software development approaches and tools, the growing complexity in software applications, and the fast ship cycles of a software plus services world, we will need to advance the state of the art in software testing. Advances in the state of the art will require experience-driven improvements that change the way testers approach testing, and how product teams approach quality. But how can we do this when the testers in most organizations are far junior in experience to their development counterparts?

A relatively small group of highly experienced senior testers at Microsoft are working to do their part of advancing the state of the art in testing. They have solved problems that have enabled entire organizations to improve testing efforts and product quality. They've done it by leveraging years of experience in software testing and the respect of their peers in order to solve extremely difficult testing problems that enabled the success of their teams.

This paper discusses three case studies showing how highly experienced testers solved big problems at Microsoft and helped their teams make improvements on quality that absolutely could not have been made without their efforts, as well as the effects a growing group of highly experienced testers can have on an organization. The paper also shows how an organization can model and groom newer testers for taking on such roles in the future, and discusses what Microsoft is doing along these same lines, and what the future plans are for leveraging senior test talent.

Software testing is a new profession compared to software development, and the roles of testers and the scope of their impact on product quality are ripe with confusion and controversy. For some, software testing is an activity performed by people whose role is to mimic the user (or in some cases by users themselves) in order to understand and report on issues that will affect the end-user usability of the software program. For example, a bank may employ a software development team to write account management software for bank tellers, and use a subset of those tellers to “test” the software before deploying throughout their entire infrastructure. For others, software testing may look beyond user scenarios and attempt to perform a more holistic evaluation of the software under test that also takes a deeper look at functional and structural aspects of the software system.

On many software teams, test roles are less valued than development roles. Indeed, if the traditional “cost of change” curve is true (see figure 1), then the emphasis should be on finding and fixing issues introduced during requirements, design, and coding – but on many test teams, the bulk of bugs are discovered much later in the software lifecycle. Despite the increased cost of fixing issues found during system test, the types of issues found during late cycle testing may not seem to warrant the same respect and pay scale as core development work does.

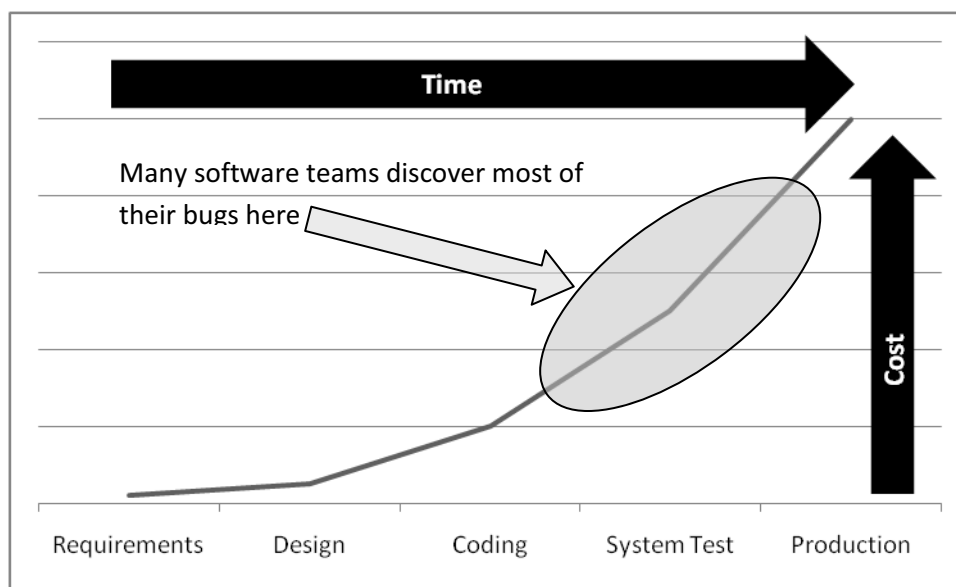


Figure 1 - Cost of change curve¹

Moving quality up stream is much more than hiring testers earlier in the product cycle. As is well known and generally accepted in the industry if you want to find errors earlier, you need to *look* for them earlier. For most software applications, finding and detecting design issues – especially during their introduction in the design phase, is a more difficult task than finding bugs during testing. Iterative methods may flatten the curve by cycling through code and test phases more quickly, but frequent iterations still often neglect to find fundamental design issues. Regardless of the specifics of the approach, a process that heavily leverages testing and quality throughout the product development cycle is a path towards better software.

¹ Boehm, Dr. Barry, Software Engineering Economics, Prentice Hall, 1981

The Microsoft Approach

For at least five years, the primary candidate pool for software testers at Microsoft has consisted of graduates from computer science related fields. While we do expect our testers to be able to write tools and automation when needed, a more complete reason why we hire from this pool is that we expect our testers to debug, diagnose, and analyze problems they run into. We also expect them to recognize patterns of bugs, and have insight into how the computer may be using the data they input. These skills decrease or eliminate the amount of time it takes to get bugs fixed (developers often know everything they need to know to fix the bug from the report), and that usually ends up increasing overall productivity. To be fair, we have had great testers without a computer science background who have been able to develop these skills successfully, but we've found much more success in hiring computer science majors and letting them develop into fantastic software testers.

At Microsoft, employees have the option of changing groups and disciplines at almost any time. Given the strong software development background of these testers, some choose to move on to software development engineer (SDE) roles at Microsoft, but most remain in test positions throughout their Microsoft careers.

Career Paths for Testers

An important thing to note is that testers at Microsoft have the same salary structure and promotion possibilities as their counterparts in other engineering disciplines. Over the last two years, a number of senior leaders in software testing at Microsoft have been studying the role of senior testers with the goal of helping to clarify the test career path. One output of this work was the creation of four senior test personas² (used internally and shared externally). Each persona describes a potential role a senior tester may play on his or her team. A common theme among the roles described in the personas is leadership. Senior testers provide technical leadership, mentoring and vision for their test teams, and they improve their test teams through this leadership. The current phase in the ongoing investigation into the roles of senior testers is to examine the roles of some of Microsoft's most senior testers and understand not only how they make their teams better, but also how they make their products better.

There has been some success in the work so far. Due partially to consistent hiring practices, and partially due to the work of test leadership, the number of testers in senior positions has grown by over 300% in the last two years. As the number of people in these senior roles has grown, it's important to be able to take a step back and ask, "To what end? Do we just have more people in senior roles, or are they collectively making an impact on product quality?" Measuring the direct impact of the increased population of senior testers is difficult, but we plan to continue investigating this impact over the coming months and years. For example, we are looking at post-release quality measures (e.g. patches, customer feedback, and data from product support), and looking for correlation between perceived quality for certain products (or product areas) and the makeup of tester experience on those product teams.

² <http://msdn.microsoft.com/en-us/testing/bb414765.aspx>

The author separately interviewed several of Microsoft's most senior testers and their managers in order to obtain qualitative information on the influence of these testers on product quality. The focus of these interviews was on testers in senior non-management roles. Their peers in a development organization are typically Software Architects or Principal Software Development Engineers.

Case Studies / Interviews

Two primary factors helped drive the basis for inclusion in these interviews. First, the testers profiled are in the career stage known inside Microsoft as "Principal". Employees at the Principal career stage comprise approximately the top 4% of all Microsoft engineers, and the top .5% of testers in non-management roles. The second factor influencing the selection of case study candidates was to target testers who most typically represent the role towards which Microsoft expects highly experienced testers to grow. Those selected are strong performers, embody the model of the senior test role and function, and in many ways act as role models for all testers at Microsoft.

The interviews with managers and employees were free form, but included a guiding set of questions. These included questions about:

- 1) Role on the team
- 2) Level of influence
- 3) Influence on improving testing
- 4) Impact on product quality
- 5) Estimated impact on testing and quality if the individual were *not* on the team

Interviews occurred in person in the spring of 2009. Three of the roles are profiled in the following section of this paper. Descriptions come primarily from their direct managers. Feedback from peers and colleagues are also included.

The Role of the Highly Experienced Tester

The testers profiled in the following section are in non-management roles. While there is definite need for senior roles in test management, the focus of this paper is on the role of senior testers in non-management roles.

The role and tasks of senior testers vary depending on the skills they possess and the needs of their organization. In some cases, they may be responsible for developing or architecting the test infrastructure used by the team (testers at Microsoft build and maintain most of their own test tools and infrastructure). In other cases, the role can begin to look more like a traditional quality assurance role, where senior testers are responsible for establishing and measuring the quality processes of all disciplines on the team (note that Microsoft does not currently have a quality assurance profession).

In many other cases, senior testers are deep subject matter experts for the technologies they test, or for specific testing methods such as security or performance testing. In all cases, senior testers bear some responsibility for making their teams better – through example, technical leadership, or through direct mentoring or coaching.

Lucretia – Windows Operating System

Lucretia has been at Microsoft for 12 years, and is a Principal Software Design Engineer in Test (SDET) on the Windows team. She is responsible for determining how to test a vast number of Windows operating systems configurations. The testing matrix for Windows SKUs (stock keeping units) is highly complex. Windows Vista, for example, has six primary SKUs available, but those SKUs are available in 32 and 64-bit versions and in dozens of languages. Additional variations are often made available due to regulatory requirements (e.g. offering Windows Vista in Europe without Media Player). A massive number of possible combinations of components could theoretically make up a Windows release. While Windows ends up shipping only a subset of these possible combinations, the ability to select from all possible combinations is necessary in order to respond to issues in a global market. Rather than worry about the implications of how to test numerous combinations of components across Windows, the work Lucretia does takes out any guesswork about what we ship to customers. For example, if removal of a feature or component from Windows is required, Lucretia's work can point out exactly which binaries need to be removed – verifying both that the feature is removed, and that *nothing else is broken* due to dependencies on the removed components. The same rules of dependency analysis are also applicable to patches or hot-fixes.

Her level of influence across all disciplines in the organization is very high. There are teams who base their strategy and workflow on the work she produces. If she were to stop doing her work, the ability to ship a hot-fix would be greatly reduced, and Microsoft's ability to react to high priority customer issues would be greatly impacted. This ability to identify and reduce risk in software is a big part of the test role at Microsoft and this core ability is nurtured and fostered throughout the career of all testers. Software developers will often come to Lucretia and ask for her opinion on how to handle specific issues related to componentization and dependency analysis. Her impact reaches far enough beyond her team to the extent that other organizations have used their own budgets to give her additional rewards. In her manager's estimation, Lucretia has found or prevented thousands of bugs. She is highly trusted and helps build many positive relationships for her manager.

Without her on the team, the cost of testing would rise dramatically. Her expertise, along with the toolset she has developed enables her to accomplish alone in 4-5 hours what used to take a 3-person team an entire week. She enables the team to ship far more SKUs, patches, and service packs than they could without her experience and knowledge of the problem domain.

Ed – SQL Server

Ed is a Principal Test Architect ("Test Architect" is a role of some Principal SDETs at Microsoft) on the SQL Server Team. Ed is deeply involved in identifying and solving the problems of his product unit, but he also has the freedom to help and educate others outside of his product group, and is extremely influential in advancing the engineering of testing across the entire company.

Recently, Ed helped a team move to a declarative testing model for some of their user interface automation scenarios – moving the team from focusing on implementation details to thinking about the scenarios that drive the test cases. This reduced complexity of verification improved efficiency on the test team and enabled the team to reach levels of test coverage equal to their previous efforts.

Ed is extremely influential in the design of tests across his entire organization. He makes the testers on the team better both through technical leadership, and by teaching other testers to think clearly for themselves and ask the right questions upfront. His approaches to test and test design have enabled the testers around him to write fewer tests yet reach higher levels of feature and code coverage.

Mentoring other testers is a big part of Ed's role. He teaches testers to approach the testing task in a structured and practical manner - making them *think* about the task they are facing and what the full picture of accomplishing that task includes before encouraging them to dive into the technical aspects of the problem. He also is astute at identifying missing subtleties in existing approaches, and constantly helps the testers on his team improve the way they think about testing.

Without Ed on the team, the momentum of advancement in his testing organization would slow measurably. It is also quite conceivable that his business unit would have done less product development, as Ed's approaches have enabled the test team to do more work with fewer people and reach higher quality. He helps testing focus on the areas that are most important.

Jim – Internet Explorer

Jim is a Principal SDET on the Internet Explorer team, and has been at Microsoft for 13 years. He primarily works on site compatibility features in Internet Explorer. He develops systems for identifying the extent and severity of compatibility issues. Although he has no direct reports, he essentially runs a large part of the testing organization through technical leadership and vision. He has created several case studies for the team (and other teams) from which to learn. For example, he would start with a bug report, and then show the team how to tear it apart and find the root issue. Currently, Jim is a big part of the planning for the next version of Internet Explorer.

He worked with many of the functional teams within the organization, and has helped them make adjustments in order to help them test better in areas such as performance and reliability. His peers often cite his experience and technical depth as a key part of his success. In at least two instances, testers were considering leaving the team to take on different roles, but in fact decided to stay due to Jim's leadership and his position as a role model.

Of all the individuals on the test team, he has had the most impact on product quality. Jim personally investigated over 1000 compatibility issues over the course of the release. Some of the issues came back as fundamental design issues in the product – issues that may not have been found at all without his work. Overall, his investigations and analysis have been critical to the success of the project.

In one example, Jim investigated an issue where menus were not displaying correctly on a web page. After extensive investigation, he discovered that the error occurred due to a design change in the application of DHTML filters when rendering web pages. He then discovered a similar bug in the product bug database initially resolved as "won't fix / by design". With Jim's added debugging information, as well as a real world example, the bug was re-opened, the design was corrected, and in the process, dozens of other rendering problems in prominent sites were fixed.

Jim is also involved in spreading his knowledge across the team to help them increase their own capabilities. Jim creates full write-ups documenting many of his debugging sessions and shares them with the rest of the team. This provides guidance and demonstrations of many important investigation concepts and debugging tools. As a result, the entire team is much better equipped to tackle difficult bug investigations and debugging challenges.

Without Jim on the team, improvements in his area would have been minimal over the previous release. The team would not have been able to scale to the breadth of issues (and depth of critical issues) in a methodic way. Many design decisions and the rendering engine would not be as good without him.

Affecting the Cost of Change

What happens when you have experienced testers on your team who can influence early code decisions and product design, and who can introduce preventive techniques into the development process? The data thus far indicates that the inclusion of highly experienced testers on a team can push discovery of many more bugs earlier in the software lifecycle, therefore saving money and increasing quality. The influence of the experienced tester throughout the product cycle would certainly have an impact on the shape of the curve, and it would be reasonable to expect that the curve would flatten.

Finding a bug late in the cycle is still expensive, but better design decisions early on can potentially lower the cost of post-release bugs. Additionally, more early detection and preventative techniques will greatly reduce the *number* of bugs found post-release decreasing the *overall* cost dramatically. If experienced testers are champions of early detection and prevention, and these practices become more prevalent across a product team the overall cost of change should surely go down. Figure 2 shows an alternate cost of change curve affected by the work of experienced testers.

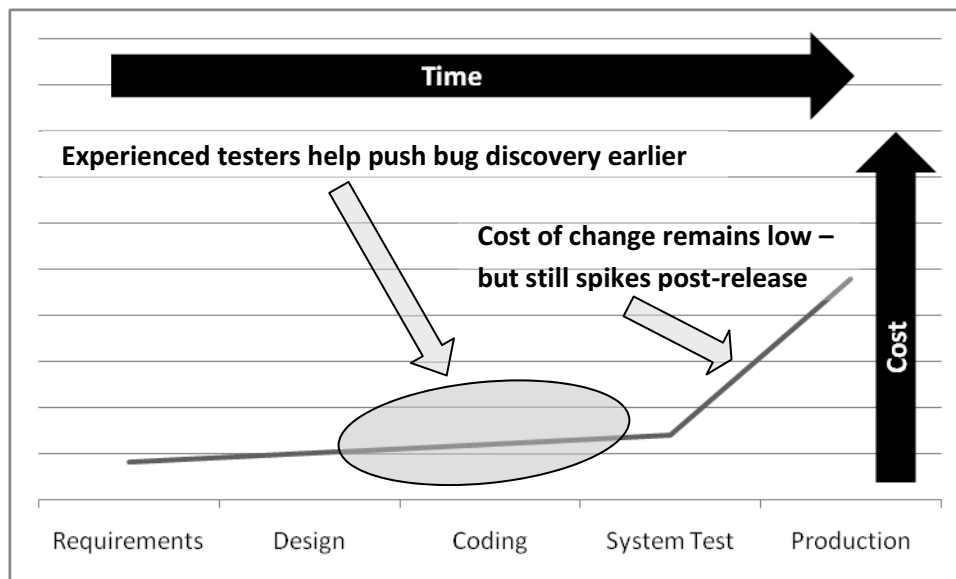


Figure 2 Alternate cost of change curve

How Do You Get There?

Testers with this level of influence and impact don't appear magically, nor are they created through special training. Testers develop into these roles through personal growth and development, as well as simply racking up years of experience shipping products along the way. Some of the common keys to success shared among nearly all of the Principal SDETs at Microsoft include:

- **Technical skills**
 - You can't be technical enough at Microsoft
- **Align your work with business goals**
 - What business problems did your efforts address today?
- **Know yourself**
 - Know the combination of skills that makes you unique
- **Be skilled and creative**
 - Continually learn and push yourself
- **Be dependable**
 - Build a reputation of reliability and confidence
- **Be productive**
 - Do your job – look for ways to be better
- **Be a brand**
 - Be an expert, be consistent, be reliable
- **Help others**
 - Look for opportunities to mentor, coach, and lead. Base your success on the success of those around you.

The above, of course, is not a recipe. Just because someone has experience, is dependable, and provides technical leadership and mentoring for his or her team doesn't mean (s)he will automatically provide the same level of impact as the individuals discussed above.

Another important point to note regarding these roles is that senior testers cannot be the saviors or heroes of their teams. They need to leverage their deep knowledge and experience to solve huge problems on their teams, but the only way for them to scale is to ensure that they spend some portion of their time helping their team get better. It is of little value to have a tester on the team whose primary role is to fight fires and take on challenges independently. Instead, they need to involve the team in order to solve problems collaboratively and give everyone on the team a chance to gain their own valuable experiences.

Organizational Structure

Organizationally, highly experienced testers often are peers of the test management team. Figure 3 shows one possible organizational structure (in this organization, the highly experienced tester has the "Test Architect" title). The experienced tester in this case has influence and leadership responsibilities for the entire organization.

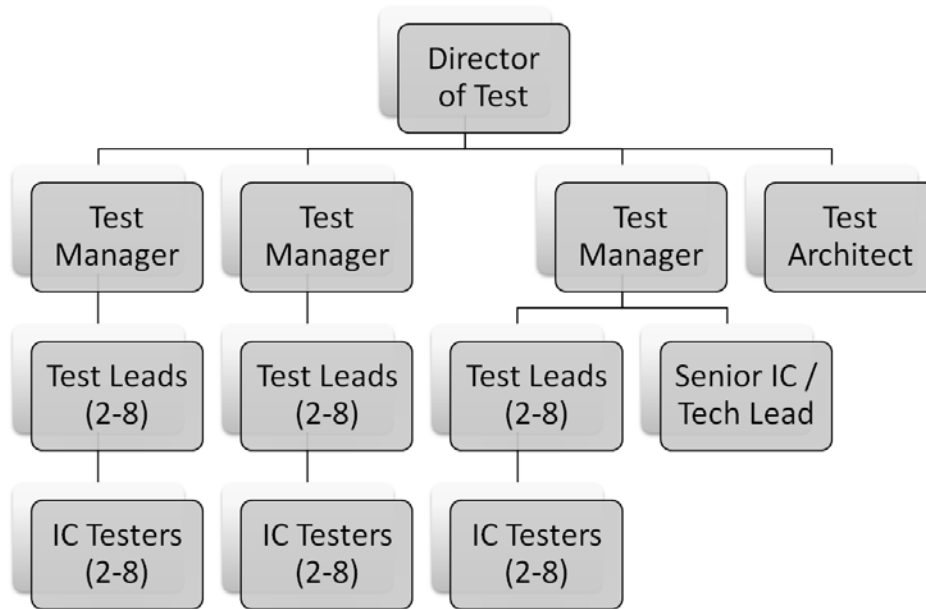


Figure 3 – Example Organizational structure

Figure 3 also shows where a Technical Lead role may be used both as a leadership role for a smaller part of the team, and as a growth path for testers who aspire to grow to a higher level of leadership and influence.

Note that this structure is common in larger organizations (30-40 or more testers). Smaller organizations may not have an individual with enough experience to take on a pivotal technical leadership position within the organization, but they often do have an individual who takes on some of the technical leadership duties in addition to her assigned testing tasks. At Microsoft,

most (but definitely not all) of the Principal SDET's work in large organizations with dozens – and sometimes hundreds of testers.

Hiring and Retention

Most teams would love to have a Jim, Lucretia, or Ed on their test team. Hiring people like these three is, however, nearly impossible. Testers like these become who they are through years of experience and challenging tasks. The role of management in creating testers like these is critical. First, testers need a proper mix of challenging tasks – work that stretches them beyond their current capabilities, balanced with work appropriate for their level of experience so that they can achieve success in their day-to-day work. It's the role of the manager to ensure the proper mix for everyone on the team. Remember that the day-to-day work for one tester may be the stretch assignment that another tester on the team needs.

Experienced testers also need opportunities for leadership. One thing in common with all three of the testers in this paper is that they all have led technical teams. Jim spent some time in his career as a manager, but in all three cases, leading “virtual” teams, e.g. leading a cross-group team in order to solve a common problem for all teams, or solving a problem for an entire test team by working with everyone on the team are necessary and typical examples of leadership exhibited by growing testers.

Finally, these experienced testers have a high degree of trust from their management chain. They're often asked to solve an extremely difficult problem for their organization, but have freedom to experiment when necessary and to choose *how* they solve the problem. Their managers know that they will ask questions or get feedback when they need it, but they also know that they will confidently make most of the decisions on their own, or with the virtual teams they build to help solve the problem.

The traits above are just as crucial for growing the next generation of highly experienced testers as they are for retaining the same. I don't know many people who would leave a job where they were valued, challenged, and trusted.

Conclusions and Next Steps

The influence of these three individuals on product quality is notable. The contributions of each one of these three have enabled their teams to accomplish much more than they could have without them. Additionally, they are leaders for their teams and are influential in helping to grow the senior testers of tomorrow. They all make their teams better through technical leadership, mentoring, and strong vision.

The data thus far is largely subjective, but planning is underway for more in depth case studies. The value of anecdotal data such as described previously in this paper is already having an impact across the company. At a grassroots level, the stories of roles of senior testers are already inspiring more people to stay in test positions. From the other side of the spectrum at the executive management level, there is a growing understanding of the types of value that experienced testers bring to an organization and there is an increasing desire to groom more testers to rise into these positions within organizations and sub-organizations across the company.

The impact of the three testers profiled above is undeniable, yet testers at their level represent just a fraction of a percent of the test population at Microsoft. As more testers, both at Microsoft and in the industry overall achieve extensive experience and technical knowledge, I expect the impact on product quality to become more and more significant. I believe it is just a matter of time before testers such as these become some of the most respected (and sought-after) members of the entire software ecosystem.

Building Alliances

Karen Johnson

This presentation looks at how to create and foster successful professional relationships with teams and people. An alliance goes beyond building a relationship. Alliances are about building reciprocal relationships. Alliances look to not just foster a collaborative spirit but also look for ways that teams can look out for each other and work together. Come and listen to tips and techniques to build relationships. Karen will specifically discuss:

- How to foster collaboration and keep communication in sync across teams.
- How to look for opportunities in work deliverables that build reciprocal alliances.
- How to find ways to keep team tensions down at the critical end game of the development process.

***Karen Johnson** is an independent software test consultant. Karen has been involved in software testing for more than two decades. Karen has extensive test management experience. Her work often focuses on strategic planning. Most recently, her focus has been on developing a sense of community for software testers working in the area of regulated software testing. Karen is a frequently invited speaker at major conferences and has published numerous articles and recorded webcasts on software testing. She blogs about her experiences with software testing www.karennjohnson.com Karen is the co-founder of the WREST workshop www.wrestworkshop.com*

*She is currently writing a chapter for the tentatively titled **Beautiful Testing**, an O'Reilly imprint, available in late in 2009.*

Building Alliances

PNSQC 2009
Karen N. Johnson

Building Alliances, Copyright © Karen N. Johnson,
2009



Instinctive Collaboration

The most important single central fact about a free market is that no exchange takes place unless both parties benefit.
Milton Friedman

Building Alliances, Copyright © Karen N. Johnson, 2009

Building Alliances



- Alliances or Adversaries
- Collaboration or Collusion
- Beautiful Results or Ugly Realities
- Reduce Tension
- Build Reciprocal Relationships
- Foster a Collaborative Energy

Building Alliances, Copyright © Karen N. Johnson,
2009



Shared Interest

In this new wave of technology, you can't do it all yourself, you have to form alliances.
Carlos Slim Helu

Building Alliances, Copyright © Karen N. Johnson, 2009

ALLIANCES

Building Alliances, Copyright © Karen N. Johnson,
2009

COLLABORATION

Building Alliances, Copyright © Karen N. Johnson,
2009



Conducive atmosphere

Collaboration equals innovation.
Michael Dell

Building Alliances, Copyright © Karen N. Johnson, 2009



Joy

We choose our joys and sorrows long before we
experience them.
Kahlil Gibran

Building Alliances, Copyright © Karen N. Johnson, 2009



Teamwork

Alone we can do so little; together we can do so much.
Helen Keller

Building Alliances, Copyright © Karen N. Johnson, 2009



Balance

Live a balanced life - learn some and think some and draw and
paint and sing and dance and play and work every day some.
Robert Fulghum

Building Alliances, Copyright © Karen N. Johnson, 2009

BEAUTIFUL RESULTS

Building Alliances, Copyright © Karen N. Johnson,
2009

ADVERSARIES

Building Alliances, Copyright © Karen N. Johnson,
2009



Secret Battles

Never yield to force; never yield to the apparently overwhelming might of the enemy.
Winston Churchill

Building Alliances, Copyright © Karen N. Johnson, 2009



Gossip, Secrets, and Lies

In an age of universal deceit, telling the truth is a revolutionary act.
George Orwell

Building Alliances, Copyright © Karen N. Johnson, 2009



Open Battles

You have enemies? Good. That means you've stood up for something, sometime in your life.
Winston Churchill

Building Alliances, Copyright © Karen N. Johnson, 2009



Email and Politics

By sending a brief memo to a colleague or a superior, you have a paper trail that Backstabber's can't erase.
Les Parrott

Building Alliances, Copyright © Karen N. Johnson, 2009

COLLUSION

Building Alliances, Copyright © Karen N. Johnson, 2009

UGLY REALITIES

Building Alliances, Copyright © Karen N. Johnson, 2009



Backstabbing

Backstabbers ooze anger.
Les Parrott

Building Alliances, Copyright © Karen N. Johnson, 2009



Exercise: Shared Experience

Stories are the creative conversion of life itself into a more powerful, clearer, more meaningful experience. They are the currency of human contact.
Robert McKee

Building Alliances, Copyright © Karen N. Johnson, 2009



Shunning

Shunning is a childish, primitive way of branding someone as an outsider.
Annette Simmons

Building Alliances, Copyright © Karen N. Johnson, 2009

REDUCE TENSION

Building Alliances, Copyright © Karen N. Johnson, 2009

EXERCISE

Building Alliances, Copyright © Karen N. Johnson, 2009



Cool Down

Let your soul stand cool and composed before a million universes.
Walt Whitman

Building Alliances, Copyright © Karen N. Johnson, 2009



Slow Down

The master of strategy does not appear fast.
The Book of Five Rings

Building Alliances, Copyright © Karen N. Johnson, 2009



Teams in Alignment

Coming together is a beginning; keeping together is progress;
working together is success.
Henry Ford

Building Alliances, Copyright © Karen N. Johnson, 2009



Seek Clarity

Clarity affords focus.
Thomas Leonard

Building Alliances, Copyright © Karen N. Johnson, 2009



Sphere of Influence

The secret of my influence has always been that it remained secret.
Salvador Dali

Building Alliances, Copyright © Karen N. Johnson, 2009

BUILD RECIPROCAL RELATIONSHIPS

Building Alliances, Copyright © Karen N. Johnson,
2009



Detect Change

If you're in a bad situation, don't worry it'll change. If you're in a
good situation, don't worry it'll change.
John A. Simone, Sr.

Building Alliances, Copyright © Karen N. Johnson, 2009

FOSTER A COLLABORATIVE ENERGY

Building Alliances, Copyright © Karen N. Johnson, 2009



Shared Success

No person will make a great business who wants to do it all himself or get all the credit.
Andrew Carnegie

Building Alliances, Copyright © Karen N. Johnson, 2009



Virtual Team

It shows 'us vs. them,' and I'm on the 'us' side.
Dan Quayle

Building Alliances, Copyright © Karen N. Johnson, 2009

EXERCISE

Building Alliances, Copyright © Karen N. Johnson, 2009



Focus

Success is focusing the full power of all you are on what you have a burning desire to achieve.
Wilfred Peterson

Building Alliances, Copyright © Karen N. Johnson, 2009



Exercise: The Unspoken Alignment Map

Irrational behavior can be understood. It only requires that you understand the map being used and the "survival" issues that surface on that particular map.
Annette Simmons, Territorial Games

Building Alliances, Copyright © Karen N. Johnson, 2009

Resources

Following are books and references I have found helpful:

- Territorial Games, Annette Simmons
- A Safe Place for Dangerous Truths, Annette Simmons
- A Book of Five Rings, Miyamoto Musashi
- Inspiring Others: What Really Motivates People, Duke Corporate Education
- Care Packages for the Workplace, Barbara Glanz
- Toxic Emotions at Work, Peter Frost
- In the Line of Fire: How to Handle Tough Questions...When It Counts, Jerry Weissman
- Tao Te Ching, Lao Tzu
- Working Through Language, Time, and Cultural Differences, Karen N. Johnson, <http://www.ftpress.com/articles/article.aspx?p=681403>

Building Alliances, Copyright © Karen N. Johnson,
2009

THANK YOU FOR YOUR TIME.

**CAN I ANSWER ANY
QUESTIONS?**

Karen N. Johnson
email: karen@karennjohnson.com

Building Alliances, Copyright © Karen N. Johnson,
2009

Testing IPv6 Enabled Applications

Travis Luke

Software Development Engineer in Test

Microsoft Corporation

tluke@microsoft.com

Abstract:

Each year IPv6 gains wider adoption around the world, both as a replacement for traditional TCP/IP and in side by side usage. IPv6 transition technologies are now enabled by default on many modern operating systems and applications have already begun to take advantage of them. However, very few testing guidelines exist for this emerging technology. It is important that we move quality forward in this ecosystem. First, I will present an overview of the current state of IPv6 deployment. Next, I will focus on the most popular transition technologies (6to4, Teredo and ISATAP). I will describe how each technology acts as a bridge to a full IPv6 deployment and what software developers and testers need to know about it. Then, I will describe how to build a test lab that simulates various IPv6 environments such as the home, the Internet café, and the enterprise. Last, and most importantly, I will outline practical test cases for testers of IPv6 enabled applications.

Author Bio:

Travis Luke has been working at Microsoft a Software Development Engineer in Test for ten years. For the past five years, he has worked in the Windows Networking division on Peer-To-Peer network technologies. Prior to Microsoft, Travis worked as a Network Administrator and consultant. He enjoys talking about himself in the 3rd person.

Why should we all care about IPv6?

There seems to be a never ending debate about IPv6. Is it really needed? Is IPv6 the solution to IPv4 address depletion? Why can't we all use a NAT and be happy? (For a definition of what a NAT is see the appendix). If you are interested in adding to this debate then there is a big world on the Internet for you. The fact of the matter is IPv6 is already here. Comcast recently announced that some residential customers will have IPv6 connectivity by 2010. Some of the most popular web sites on the Internet now have IPv6 offerings. (ipv6.netflix.com, ipv6.Google.com). The 2008 Olympics featured an official IPv6 enabled website (ipv6.beijing2008.cn/en). In fact all network operations of the Olympic Games that year were conducted using IPv6. Even the Pirate Bay has IPv6 enabled web sites and trackers (ipv6.thepiratebay.org). Microsoft Windows Vista and Apple Mac OS X v10.3 support IPv6 and is enabled by default.

There is a wide range of applications using IPv6. Of course, the obvious applications are the web browsers. Since the World Wide Web is the biggest killer app for IPv4, it makes sense that it will be for IPv6 as well. However, there is another class of applications that use IPv6 for its ability to traverse NATs and tunnel over IPv4. Examples of this include Videoconferencing tools such as Ekiga/Gnomemeetin, and ISABEL. Other examples are Media streaming applications like Windows Media Player and Videolan. This may also include games such as Quake3, or peer to peer file sharing applications such as Gnuetella. The Remote Assistance tool in Windows 7 uses IPv6 tunneled through IPv4 to establish end-to-end connections with your peers to request or offer support.

So it is clear that IPv6 is something we should all care about. Chances are we have used it without realizing it. Therefore, as we build applications it becomes important to test that they work well with IPv6 and provide a seamless transition to the end user.

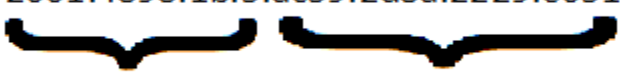
The Five-Minute IPv6 Refresher Course

This is a brief overview of IPv6. This will help build the context for the discussion later in this paper. Many important details and concepts will be skipped over. For a complete overview of IPv6 there are

A Link Local Address

fe80::205:ddff:fe27:3840

many books available. See the bibliography at the end of this paper for references to my favorites. An IPv6 Address is a 128-bit address that is printed in hexadecimal format. Each 16-bit block is separated by a colon. For example the address on my PC is 2001:4898:1b:5:ac39:2d8a:2229:e051. There is a shortcut to compress the zeros by using two consecutive colons to indicate 16-bit blocks comprised of zeros. For example fe80:0:0:0:205:ddff:fe27:3840 is the same as fe80::205:ddff:fe27:3840. Addresses come in three types: Unicast, Multicast, and Anycast. For the purposes of this discussion we will focus on Unicast addresses. The first 48-bits (3 blocks) represent the Global Routing Prefix. This distinguishes the scope of the address. Global addresses are addresses that can be reached by any other global address. This is

2001:4898:1b:5:ac39:2d8a:2229:e051

Network ID Host ID

similar to the public addresses range on IPv4 networks. The next 16 bits represent the subnet ID. Link-local addresses are not routable and are only reachable by nodes on the same subnet. Link-local addresses always have the Global Routing Prefix and subnet ID of fe80:0:0:0. Together the Global Routing Prefix and the subnet ID make up the network ID of the address.

The remaining 64 bits is the interface ID. Your interface ID is assigned by the network stack. The MAC address of your adapter is often used to create the interface ID. Each adapter you have is given a link-local address by the network stack. If your router is configured for IPv6 it will assign your adapter a Global subnet ID and your network stack will create one or more Global IPv6 addresses. In this way, each adapter may have many addresses. Unlike IPv4 the subnet ID and interface ID is fixed so there is no need for a subnet mask.

Sometimes you will see a %nn following an IPv6 address. This is called the scope ID. The number following the '%' sign is the adapter ID that this address is associated with. Usually scope IDs are only used in link-local addresses.

IPv6 Transition Technologies

To help ease migration to IPv6, a number of transition technologies have developed. Each of these helps in its own way and has its own quirks.

Intra-Site Automatic Tunnel Addressing Protocol (ISATAP)

ISATAP creates IPv6 addresses based on IPv4 addresses. An ISATAP address looks like this:

2001:4898:0:fff:200:5efe:157.59.29.11. An ISATAP address can have a link-local or a global scope. Link-

local addresses are limited to connectivity to nodes on the same subnet. Global Scope ISATAP addresses can go anywhere that the router lets them go. To get a Global ISATAP address there must be a ISATAP server in the network that is configured to assign addresses, and you must have routes defined to route the addresses across subnets. Due to these limitations ISATAP is typically deployed in Enterprise Scenarios. ISATAP addresses can be identified by looking at the first 32-bits of the host ID. They always have a starting host ID of 0:5efe or 200:5efe.

A Link Local ISATAP address

fe80::200:5efe:157.59.29.11

6to4

This protocol allows IPv6 packets to be transmitted over a IPv4 network without the need to create an explicit tunnel. The Global 6to4 IPv6 address is generated based on the public IPv4 address. The upside of this is that if

you have a public IPv4 address you can instantly have access to a 6to4 IPv6 address and fully access IPv6 resources. The downside is you must have a public IPv4 address. If you are behind a NAT then most likely your NAT hides the public IPv4 address and assigns you a private ipv4

A 6to4 Address

2002:836b:1::836b:1

address. However some modern NATs such as the Apple AirPort Extreme, actually assign 6to4 addresses to all of its connected nodes. 6to4 addresses can be identified with the first 16 bits of the network ID which are always 2002.

Teredo

Teredo is a tunneling protocol that grants Global IPv6 connectivity through NATs. It works by connecting to a

A Teredo Address

2001::cd49:7601:a866:efff:62c3:fffe

Teredo Server on the internet using UDP

over IPv4. That server assigns a Global IPv6 address to the Teredo Client. Then the server routes traffic between that client and other Teredo and Non Teredo hosts. Teredo is not compatible with all NAT devices. Teredo has become very popular not just for IPv6 connectivity, but also as a convenient way to seamlessly connect to nodes behind NATs. In fact, the name Teredo is the Latin name for shipworm which is a reference to the protocol's ability to "punch holes" through NATs. Teredo addresses can be identified by the first 32 bits of the network ID which are always 2001:0.

Building your Test Infrastructure

In order to test your applications on IPv6 networks you need to create an IPv6 network. However as you can see from the many transition technologies above, there are many different types of deployments. The first step to building your test lab is to identify who your customer is and where they will be running your software. Then you can understand your customer's environment and scenarios. Let's analyze the home, public, and enterprise network topologies.

Today's home network usually consists of one high speed Internet connection connected to a wireless NAT, which connects to one or more PCs. However, not all home networks are created equal. Some do not have a NAT and get their connection straight from the ISP. And not all NATs are created equal. There are various NAT algorithms in use such as Cone Nats, Port restricted NATs, and Symmetric NATs. The latter, symmetric NATs, may inhibit Teredo functionality. Some NATs natively support 6to4. Others have firewalls that block all IPv6 traffic. To make matters worse, some NATs have been known to crash or hang in the presence of some specific IPv6 traffic. So it is important, when you test your home scenario, that you use a wide variety of NAT hardware and settings. Most likely you will not be able to solve all problems related to home networking hardware and configurations. But as a software developer, you will be able to discover how your software behaves in the presence of a hostile networking ecosystem. What you learn can then be applied to product support, recommended hardware guides, and documentation.

Network Environment	IPv6 type
No Internet Access	Link-local IPv6
Dialup	6to4
Direct Connection to Public Internet	6to4
NAT (non-6to4)	Teredo / Link-local IPv6
NAT (6to4)	6to4 / Link-local IPv6

Public networks are in many ways similar to home networks; however there are a few things to be aware of. Administrators of public networks want to ensure safe Internet access for its users. Therefore, the firewall settings on public networks tend to block more traffic than usual. Because of this, Teredo functionality can be limited. Some public networks take the additional precaution of blocking traffic between peers connected on the same network. Many public networks require you to go to a login web-site and accept their terms of usage before your traffic is routed to the public Internet. This timing may affect the Teredo Service's ability to connect to the Teredo server. Also, some public networks are configured incorrectly, which breaks IPv6 connectivity. For example, you may occasionally encounter a public network that assigns public IPv4 addresses to all of its clients, but is still behind a NAT. This will make the clients assign themselves 6to4 addresses but be unable to connect to IPv6 resources outside of the NAT. If you are building an application that will be used in public networks you may want to build a test lab that simulates these types of problems. Additionally you may want to visit public networks in your local area and test it out yourself in the real world.

Network Environment	IPv6 type
Basic Internet Access	Teredo / Link-local IPv6

Enterprise Networks are the easiest to simulate in a lab. This is because enterprises typically provide greater connectivity between all nodes. Most enterprises use proxy servers which completely limit external Global IPv6 access. Increasingly Enterprises are deploying either Native IPv6 internally or configuring an ISATAP server. There may be challenges involving remote access however. Not all remote access hardware and software is IPv6 aware. Interestingly, Windows 7 "Direct Access" technology is based on Teredo.

Network Environment	IPv6 type
IPv4 Only	Link-local IPv6 / Link-local ISATAP
ISATAP Servers	Link-local IPv6 / Link-local ISATAP/Global ISATAP
Native IPv6	Link-local IPv6 / Native Global IPv6
Remote Access	? Depends on method of remote access

Test Cases for your IPv6 Enabled Applications

There are a number of test cases that should be executed against all IPv6-enabled applications.

1. Verify proper address selection.

With IPv6 there are many addresses associated with a single adapter. Your application must choose to bind or advertise the correct address.

- a. *Is the application choosing an address from the correct scope?*

If you want resources outside of your subnet to connect to your application you need to make sure that you bind to and advertise a Global IPv6 address. If you are only going to communicate with machines on the same subnet then you need to verify that it chooses addresses on the link-local scope. You also need to verify that your application performs the correct action in the event the address scope you want is not available.

- b. *Does the application properly react to address changes?*

Sometimes the router or transition technology is slow about assigning a global address, so your application may need to listen for and react to address changes. You may need to simulate address changes by adding routes to your router or enabling/disabling an upstream Internet connection.

- c. *Is the application choosing the best address within the selected scope?*

It is possible that you have multiple Global IPv6 addresses. You could have a Teredo address and a 6to4 address. You may have a Native IPv6 address and a ISATAP address. You need to verify that your application listens to or advertises the best address for its needs.

- d. *Is my application using the Public or Anonymous Address?*

When dealing with Native IPv6, the networking stack will often assign a second Global IPv6 address. This is called an anonymous or temporary address. This address has the same network ID as the first address but with a different host ID. While the Host ID for the first address is based on your MAC address, the Host ID for the anonymous address is random, and changes every few hours. The rationale behind this is to allow for anonymous access to Internet resources. If you connected with your public address while browsing the Internet web sites could track your usage since they could derive your MAC address from the address. However if you connect with your anonymous address they can only track you until your address changes. So in some cases an application should use the anonymous address. In other cases, primarily involving server scenarios requiring fixed, unchanging addresses, you must use the public address.

2. Verify the application works in IPv6 only environments

This means configuring an environment in which there is no IPv4, and disabling IPv4 in your operating system. If your application is designed to run in Native IPv6 only scenarios then disable IPv4 and make sure that you don't have any hidden dependencies. Often times networking applications make use of protocols such as HTTP, ICMP, SSDP, or WSD. The application may need to invoke the correct flags or APIs to ensure that these protocols are not using IPv4.

3. Verify the application behaves when IPv6 is not available.

Disable IPv6 on your machine and try to use your application. Does it failover to IPv4? Does it crash? Or does it give a useful error message?

4. Verify security settings.

Because Teredo "punches holes through NATs", it opens up potential security risks of unsolicited Internet traffic reaching your application. To combat this, the Teredo implementation on Microsoft Windows has a strict security model that must be followed for the Teredo client

service to operate. This includes having an IPv6 capable firewall. Additionally, there are some socket options which can be set to allow or disallow unsolicited traffic from the Internet. For more information on this see the MSDN article at: <http://msdn.microsoft.com/en-us/library/aa832668.aspx>.

5. Test with many personal firewall vendors.

Many PCs come bundled with personal firewalls. Not all of these firewalls have been fully tested with IPv6. Some firewalls block all IPv6 traffic. Some allow all IPv6 traffic unfiltered. By testing on a variety of the most popular firewalls you can add to your knowledge base how your software reacts. This knowledge can be added to your documentation, support library, or recommended software/hardware lists.

6. Do not share addresses with the scope ID.

Link-local addresses often have the scope ID at the end to indicate the adapter associated with that address. It is written in the form of %nn. For example fe80::ac39:2d8a:2229:e051%13 has a scope ID of 13. The scope ID is only useful for the machine that the address is on. If you convert that address to a string, copy it to another computer, and try to ping that address the scope ID would be meaningless or misleading. So if your application ever shares the address with another machine you must verify that the scope ID is removed.

7. Correctly format any address displayed.

Whenever an IPv6 address is displayed it should be correctly formatted. These are not major blocking issues. They are cosmetic bugs. IPv6 addresses are large and hard to read. By following these guidelines it will be easier to read the addresses. When used in application tracing and test logs it will be easier to spot other bugs. In general you may want to avoid displaying addresses to the user. When you must display them then it is important to follow these guidelines. Most IPv6 implementations come with an API that helps in properly formatting an address for displaying. The following table shows common bugs with displayed addresses:

Wrong Format	Correct Format	Comments
fe80:0:0:0:ac39:2d8a:2229:e051%13	fe80::ac39:2d8a:2229:e051%13	Use the :: shortcut to cut out octets with a zero value
2001:4898:001b:0005:ac39:2d8a:2229:e051	2001:4898:1b:5:ac39:2d8a:2229:e051	Do not display leading zeros in octets
fe80::200:fefe:9d3b:1d0b	fe80::200:5efe:157.59.29.11	ISATAP addresses should display the embedded IPv4 address rather than the hexadecimal equivalent.

Happy Testing

IPv6 is a technology which we have to face. As more applications begin to taking advantage of the end-to-end connectivity and ISP begin IPv6 offerings the need for support will increase. The wide variety of IPv6 transition technologies makes testing difficult because of the wide amount of test environments. By focusing on your primary customer scenarios, test coverage becomes achievable. The test cases outlined in this document will help ensure quality in your product as the transition to IPv6 moves forward. More than anything a good background knowledge of IPv6 and your customer scenarios is the key to success.

Appendix

What is a NAT?

NAT stands for Network Address Translation. It is a device that acts as a router between the Internet and a private network. Nearly all home wireless routers serve as a NAT. On the private side of the NAT, or the home network, devices are assigned addresses private IPv4 addresses. These addresses are not routable by devices with public IPv4 address which reside on the Internet side of the NAT. The NAT software takes outgoing packets from the private network and modifies the header to make it appear it came from the NAT's public address. The NAT attempts to route incoming packets to the correct host on the private network. If it cannot determine what host the packet is intended for the NAT will drop the packet. This technology works great for web browsing and email. However it works poorly for establishing end-to-end connectivity between two hosts. Wikipedia has an excellent article on NATs at http://en.wikipedia.org/wiki/Network_address_translation.

Bibliography

Davies, Joseph. Understanding IPv6, Second Edition. Redmond WA: Microsoft Press, 2008

Hagen, Silvia. IPv6 Essentials, Second Edition. Sebastopol, CA: O'Reilly Media, 2006

Blanchet, Marc. Migrating to IPv6. West Sussex, England: John Wiley & Sons Ltd, 2006

Kerner, Sean Michael. "Comcast Embraces IPv6" 18 June 2009 <
<http://www.internetnews.com/infra/article.phpr/3825696/Comcast+Embraces+IPv6.htm>>

Test Faster

John Ruberto
Software Quality Leader, QuickBooks Online
Intuit, Inc.

John_Ruberto@Intuit.com

Abstract

Moving Quality Forward can be a daunting task. Making changes to the status quo is already difficult, and that is before you consider the complexity and interconnectedness of your existing practices. When pressed to make changes, it can seem like an overwhelming proposition.

This paper will show how our team responded to such a challenge: to cut the system test duration from 12 weeks down to 4 weeks. When initially presented with this challenge, we resisted. Our brainstorming sessions turned into justifications of the status quo and explanations of why it's an impossible task. We've always needed every minute of those 12 weeks, so of course it was an impossible task to remove 67% of the time.

The break through came when we built a model of the time we spent in the system test phase. That model allowed us to break the large, complex problem into a series of smaller, easier to implement solutions. The model we developed expressed the testing duration as a function of:

- Number of test cycles
- Number of test cases
- Rate which we could execute manual tests
- Number of defects we will find and handle
- Rate which we handle the defects
- Number of people on our test team

Now, instead of racking our brains on how to reduce the duration from 12 weeks, we could focus on a single variable at a time. For example, "how can we reduce the number of test cases executed, while maintaining the same quality levels?" Asking these questions were more productive, and lead to a number of changes that we implemented.

In the end, we succeeded in achieving a 4 week system test cycle, which enabled our company to release 5 versions of our product each year, instead of 1 release every 9 months. This improvement resulted in delivering more value to our customers, faster.

This paper will first provide some context about the industry, process, and business situation. Then, will show how we created the model based on our processes and practices, and how we used the model to create a series of smaller & easier problems to solve. The paper will then briefly describe the techniques utilized.

Biography

John has been developing software in a variety of roles for 23 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 Software Quality Leader for QuickBooks Online for Intuit, Inc. 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.

Introduction

This paper tells the story of a test team which was challenged to change due to changing business conditions. The challenge was to reduce the system test duration from 12 weeks down to 4 weeks. The complexity and difficulty of this change became apparent each time the question came up. Responses were overwhelmingly weighted towards explanations on why it was impossible, and defenses of the status quo.

The team did succeed in reducing the system test duration. The break though started happening when we started breaking down the difficult & complex (i.e. “impossible”) problem into simpler problems. There was still a lot of hard work ahead, but the team was able to move forward to implement specific changes that allowed them to meet the goal.

This paper will first provide some context about the industry, process, and business situation. Then, will show how we created the model of test duration based on our processes, and how we used the model to create a series of smaller & easier problems to solve. Finally, the paper will briefly describe each of the techniques utilized.

Background

The company is a major supplier of telecommunications equipment in the United States, supplying to all of the major telecommunications service providers. The telecommunications equipment provides the “last mile” of connection, from the Telco central office to the home or office. The equipment provides both Plain Old Telephone Service (POTS) and high speed Internet through Asymmetric Digital Subscriber Line (ADSL).

Being a critical link in the telephone service, which provides dial tone and 911 emergency access, the industry practices are relatively strict for quality and validation of the quality. The processes used to build and verify the software had been reviewed and approved by standards setting bodies, and compliance is regularly audited.

The software for this project was a management system, providing features like service provisioning, maintenance, and system alerts. It's has a client/server architecture, built on a Unix platform, and written in C++. The typical project length was 9 months, roughly 6 months for development, and 3 months for System Verification Test.

The process used for verification was built upon 3 verification cycles after features were completed. The goal for the first cycle is to execute 100% of the available tests, in theory exposing all defects. Bug fixing is allowed during the first cycle. The second cycle is reduced in size, the goal to execute approximately 50% of the tests, with the main goal to verify the bug fixes and ensure no regressions occurred. The third cycle is a sanity test on the release build.

The product existed for several years with roughly the same 9 month delivery life cycle when a business change caused an acceleration of the development. Customers want to manage more equipment types with the same software, so this extra content was built in. Since this was a management platform, release was necessarily synchronized with the hardware delivery. Each equipment type had its own delivery schedule, resulting in many more releases for the management system. Over an 18-month period, the required life cycle went from 9 months, to 5, down to 3. Leaving 4 weeks for the verification test phase.

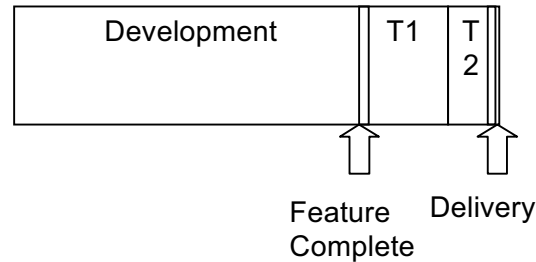


Figure 1. Development and Test Lifecycle Shows Three Test Cycles Following Feature Complete

Prior to the business change, the test duration for most projects was 12 weeks, and for all of the projects the team needed every minute of that 12 weeks, testing right up to the last minute. But, this was not a pure testing activity. Our customers were providing requirements during this time, which turned into Change Requests. Another development activity that occurred was the implementation of the database upgrade scripts and procedures, to optimize the schema migration, it was developed only after Code Complete and delivered to test 3-4 weeks afterwards.

The test team involved in this project was comprised of approximately 40 test engineers, with 10 engineers at the main development site and the balance located at an off-shore outsourcing partner. Most of these changes were driven by the managers & leaders of the test team, about 6 people in all. The skill sets and experience levels varied from Software Engineers working in quality, Quality Analysts with deep domain knowledge, and Quality Analysts with varied testing experiences. The entire journey was not an overnight success, it took approximately 2 years to reduce the test cycle duration from 12 weeks down to 4.

Model the Test Duration

Change Management Process

Initially, the desire to reduce the testing duration was presented as an aspirational goal: “Wouldn’t it be great to test in 10 weeks instead of 12?” To see if meeting the goal was feasible, we held several brainstorming sessions. These largely became gripe sessions over the impossibility of the task, it being someone else’s fault to fix, etc.

When the question was posed, “name 1 thing we spend our time on during the testing phase”. The answer was dealing with bugs. Asking, then, how can we reduce the number of bugs that we find, then some specific ideas came forth.

Model the Time and Effort

We were more productive when trying to solve a specific problem (bug reduction), rather than the complex (pull 8 weeks out of a 12 week project). We broke the whole process down to a model, with several variables. To get our minds around this, we wrote it as a mathematical equation, though it was not that precise. Here is the resulting “equation”:

$$TestDuration \approx Numberof\ Cycles \times \left[\frac{(TestCases \times TestExecutionRate + Defects \times DefectHandlingRate)}{Numberof\ Testers} \right]$$

The total test duration is a function of the number of test cycles, the number of test cases we execute, how fast we can execute those tests, the number of bugs we find along the way, how fast we can report those bugs and verify the fixes, and finally, greatly influenced by the number of testers we use to do all of this work.

Now, with this model, we can start asking more productive questions. Namely, to minimize the test duration, the following questions help guide more effective ideas:

- Do we really need to run 3 cycles?
- Can we start the first cycle early?
- How can we execute fewer test cases, while maintaining the same coverage?
- What can we do to increase the average test execution rate?
- We spend a lot of time dealing with defects? How can we reduce the defects that have to be handled?
- How can we handle each bug quicker?
- Where can we find more people to help? What is the best way to deploy new help?

These questions helped focus our thoughts and energies into productive changes.

Test Faster Tactics

This section describes many of the tactics that we ended up using to save time during the system testing phase. There wasn't a single "silver bullet", but taken together each of these tactics shaved some time off the duration.

Many of these tactics are described in greater detail in other papers and sources. This section of the paper will give a brief description of the tactic and how effective use of the tactic reduced our verification time. The references section of the paper will provide pointers to these other sources of information.

Tactic: Reduced Test Cycles

Start Regression Test Early

During one release, we ran an experiment by starting the regression test cycle early, before the Feature Complete milestone. The areas tested were focused on those that were largely complete already. We found a lot of bugs, but about the same as we would have if we waited until all the features were complete. The developers had completed approximately half of the features, and we planned the regressions around those areas.

The early start on regression testing allowed some of the developers to start fixing bugs earlier, and allowed us to deploy testers to other areas. There were a few cases where we had to completely retest areas, because they weren't fully baked when the tests were executed. But in our post-mortem, all agreed the benefit of earlier bug detection was worth the inefficiencies introduced.

Another issue to manage around this was the testers usually used this time, the few weeks before Feature Complete, to write the test plans for new features. We managed around this by having an outsourced test service provider execute these tests, while the new feature test plans were completed.

Tactic: Reduced Test Cases to Execute

Historical Analysis & Test Case Pruning

We had been building this project for a few years when presented with this challenge. While thinking about reducing the number of test cases to execute, we did a historical analysis of the test results over the various cycles. One interesting pattern, about 15% of the test cases never failed. In the history of the product, certain test cases never failed during system test. This led to the question, why execute them again?

We used this information to de-prioritize these tests, using 2 tactics. We would make sure to execute the “golden test cases” only once for the entire test duration, making sure that we didn’t use time executing them in cycles 2 and 3. The other tactic was to apply sampling for the tests in the same area. Instead of executing 150 test cases, we chose 20 at random and executed those. If all passed, then we didn’t test the rest. If any failed, we planned to execute every “golden test”, but never had to take that path.

Risk Based Testing

Related to using historical analysis, we prioritized all of the test areas based on risk, calling this Risk Based Testing. For each test area, we rated the tests in two criteria, impact to our customer if broken and likelihood of the area having problems. The ratings were high, medium, and low. For test areas that were rated with high customer impact and high likelihood of problem, we ran these first, with our best testers, and spend the most time. For the low/low tests, we followed similar procedures for the golden tests described above.

The customer impact was determined using several sources of information:

- We asked the customers what was most important to them
- The marketing & product management teams provided input
- The support team provided input
- We examined the user activity logs in the live servers.

Next, we rated the test areas on the likelihood of errors. Several of the techniques used were:

- Polling the test team showed several areas that always fail, and some that never fail.
- We mined the bug tracking system to confirm the opinions
- The code review database showed a high correlation between effective reviews and lower risk of defects
- We asked the developers, what areas were they nervous about.

Looking at this as a matrix, we would be sure to test thoroughly those areas that had the highest risk of impacting our customers. Those areas with low risk received less attention from the test team.

Customer Impact		Likelihood of Serious Issues		
		Low	Medium	High
	High	Test Thoroughly	Test Thoroughly	Test Thoroughly
	Medium	Test Moderately	Test Thoroughly	Test Thoroughly
Low	Low	Test Lightly	Test Moderately	Test Moderately
	High	Test Lightly	Test Moderately	Test Moderately

The user activity log exercise, where we examined usage in live servers, was particularly illuminating. We found that 90% of the product usage was in 3 commands: Provisioning service, logging in, and logging out. This information guided our automation efforts.

Pairwise Analysis

One driving force behind the total number of test that had to be executed was driven by a variety of possible configurations that we had to validate. We calculated that one suite, which had to vary by platform (CPU speed), memory, and a few other variables would take 4.6 billion tests to exhaustively test the complete matrix.

Another test had required 72 different permutations, between 3 OS versions, several binary state variables, and 3 different product versions. We had to reduce the set to a more manageable number of test cases.

Pairwise analysis is a method for reducing a large, exhaustive, matrix down to a much smaller set of relevant tests. [1] The theory behind this method is that most issues arise from interactions between 2 variables, not more complex combinations.

The following table shows a simple example. Suppose you had 3 binary variables, this will lead to 8 permutations for an exhaustive test. If, instead, you only executed the 4 tests selected in the table, every permutation of the 3 pairs (AB, BC, and AC) will have been tested (00, 01, 10, and 11). This cuts your test matrix in half.

Variables and Possible States			Pairs and Selected Values		
A	B	C	AB	BC	AC
0	0	0	00	00	00
0	0	1			
0	1	0			
0	1	1	01	11	01
1	0	0			
1	0	1	10	01	11
1	1	0	11	10	10
1	1	1			

In the case encountered by our team, the full matrix would have taken 72 tests to fully cover. Each test took about 4 hours to execute. Instead, by using the pairwise analysis, we were able to pare the testing down to 12 test configurations, theoretically saving 6 staff weeks of effort.

Exploratory Testing

Our team was lucky; we had a couple of senior testers that had been around the industry and this product for years. They knew the product inside and out, and had deep customer and domain knowledge. These people are priceless. They have an ability to quickly assess a feature or area, without having to be guided by written test cases. They follow their nose, using exploratory testing.

Exploratory testing is the practice of using the product with an open mind, and looking for goodness or badness. It is testing guided by the experience and intuition of testers, with exact actions determined on the fly (i.e. not following a script). The exploratory tester will be guided by knowledge of what's important for

the product, their experience, and by the observations they make while running the tests. They are exploring the product, looking for bugs.

For instance, we may ask a tester to check out a new interface for controlling the provisioned data rates for DSL service. Instead of consulting the specification or written test scripts, the tester will start with hands on activities. They may open the window, check out all the buttons and controls, visit each menu command, and start filling in forms. If the system behaves as expected, they will start looking elsewhere, or deeper into the interface. The mindset is that there are bugs in there, waiting to be discovered.

If, instead, they started by reading the specification or test plan, this may have biased their mental model on how the system should work – and they may have inadvertently skipped some promising areas or operations. Following a script constrains thinking.

Normally, these are the people we put on the difficult areas, and those rated the highest in risk. They can also reduce the overall test duration by giving an area the quick once over, instead of following a script. Frequently, 1 hour of an expert gives more valuable information than 2 days of a beginner following a script.

Another use for Exploratory Testing was when junior testers reported results that were too good to be true. A whole test suite ran perfectly, with zero failures. This sounded great, but just in case we had a senior tester give those features a quick once over. Turned out the quality was that good, and the second look increase our confidence.

The value of including exploratory testing was illustrated to me when presenting our test progress metrics to our senior leadership team. I had presented the test status, number of tests executed, number pass, and number failed. That week, we executed several hundred test cases, and had less than 5 test cases fail. Next, I presented the bug metrics, including a chart that shows the number of bugs found and number fixed that week. This chart showed 30 bugs were found.

One of the directors asked how we could have failed only 5 tests, but entered 30 bugs. I explained that the bugs were found in activities other than executing test cases. This answer solicited a follow up question about the quality of our test plans, if so many bugs “escaped” our test plans.

Thinking about this later, I came to the realization that we wrote our test cases based on the material in the specifications. The developers created their designs and code from the same set of specifications. The prepared test scripts were less “productive” in finding defects because the same thinking went into code generation as test generation. The exploratory tests found a lot of new issues because different thinking created those tests.

Tactic: Increasing the Test Execution Rate

Trimming and tuning the number of test cases had some benefit, but not nearly enough to dramatically reduce the overall duration of the test cycles. For the remaining tests, we want to increase the execution rate, which means test automation.

Test Automation

Test automation has several advantages for increasing the test execution rate.

- The obvious, computers move faster than humans, so the test steps are generally executed faster than a person can work a keyboard & verify information on a screen.

- Automated tests can also be run in parallel, running several test suites at once on different targets or clients.
- Automated tests can be triggered to run at night or on weekends. This is especially useful for build verification tests, either in a nightly build environment or for continuous integration systems.
- Automation can also be used for tasks other than actually running a test, to help the testers be more efficient. Test setup is a task that can often be very time consuming. Also, assisting testers in results analysis can save time & reduce errors.
- Automated tests can be executed more frequently than manual tests; the machines don't get bored/tired/burned out. This helps to find problems sooner, closer to the time when the problem was injected. Finding problems soon after they are introduced almost always allows the developers to fix them quicker.

A full discussion of test automation is beyond the scope of this paper. The references section provides some good starting points. Here are a few lessons we learned while instituting our test automation program:

- Start with a dedicated automation team. Balancing manual testing with starting an automation program is more prone to failure than starting with a dedicated team.
- Identify all the exploitable interfaces in the system first, before starting to automate tests.
- Exploit the machine to machine interfaces before starting UI automation. Interfaces like APIs, database access, etc. It usually easier to create tests for these interfaces, and the interfaces are less prone to changes than the UI. The resulting tests are likely to be more durable and run faster.
- Enlist help from the development team to build in testability. The development team should also be building unit and white-box tests.
- Develop automated tests using sound software development practices (design, frameworks, code reviews, test the tests, source code management, etc.). The automated tests are software too!

Test automation often suffers from the silver bullet syndrome, where its presented as a cure for all that ills software testing. Use of test automation has been a net positive in this project, but it does have its limitations. Some of the issues encountered were:

- Automated tests run very predictably, which fails to expose issues that may exist that are caused by interactions between test suites or variations of user actions.
- Inaccuracy of test results can cause your tests to be worse than worthless. False failures and missed defects will erode the confidence in the automated tests. We frequently ran the automated tests, then had to rerun them manually to confirm the results. A concerted effort to refactor the tests to eliminate these false failures fixed the problem, but the credibility of tests suffered for a while after the problems were fixed.
- Don't underestimate the effort required to maintain the tests in response to product changes.

Tactic: Fewer Defects to Manage

Out of all the topics covered in this paper, and techniques applied to reduce testing duration, simply having a better product with fewer bugs in itself is the most powerful. Doing an evaluation of all the tests that we executed, we could have executed the entire test program in 2 weeks or less, if it wasn't for the bugs. And that is without any test automation.

Because bugs exist, we have to run multiple cycles of test to retest bug fixes and verify that bug fixes did not introduce new bugs into the system. Also, multiple builds that are introduced during the test cycles, meaning downtime from testing while tests verified the build was OK, and installed on all the test servers.

Reducing the overall number of bugs is a huge reason why we were running tests anyway. Here are some of the practices that have proved effective in reducing the bugs, and increasing quality.

Effective Code Reviews

Code reviews are one of the best investments in quality. We measured the return on investment for reviews, by measuring the total effort it takes to conduct the review and comparing to the effort it would take to find the bugs by testing & fixing later in the test cycle. This return on investment ranged averaged 5:1, meaning 1 hour of effort reviewing the code saved 5 hours of finding and fixing the bugs during system test. This measurement really helped sell the idea of code reviews in the development team. We went from "there is no time to do code reviews" to "an hour of review now will save me half a days work next month".

Here are a few learning's for implementing our code review process:

- Start small. We initially tried IBM's Fagan Inspections, but found that the team became allergic to the formality and data recording. It was difficult to go from 0 to IBM in one month.
- We built a light-weight review tracking system. It was web based and easy to record the reviews. The system assigned a number to each review, so we could verify that each feature had the appropriate reviews. The tracking system had a record of the reviewers, how much time each person prepared for the review, and a list of defects found at the review. Each defect had an open/closed field to track closure.
- We created a report called the "ducks in a row" report, that listed each feature, and the tracking numbers for the design review & code review. This report was reviewed at the project reviews. The adage, "you get the behavior that you measure" helped here. An example report is shown here:

Feature	Feature #	Rqmt Review	Design Review	Code Review	Unit Test	Functional Test	System Test
8-way DSL Support	13482	#1234 3 pages, 2 defects	#1348 8 pages, 5 defects	#1532 LOC: 1354 4 defects	15 exec 15 pass	12 exec 11 pass	9 exec 9 pass

- Start measuring simply: did the review happen or not? Then, as the team became used to the process, we started looking at the reviews for effectiveness. A couple of key measures for effectiveness was the review rate (LOC/hour), and the ROI (effort saved/effort expended).

Build Quality In

Many of the problems that we had to work with during the test phase were caused by mis-understandings during the definition & development phase. Sometimes the developers mis-understood the requirements and would build the feature incorrectly, or the tester mis-understood and failed to test properly.

Including the senior test engineers from the beginning of development helped ease this process. They would participate in the requirements, design, and sometimes the code reviews. Each time they participated up front, we noticed that they were able to prevent serious bugs before the bug was actually introduced.

Use Automation to Test Constantly

Automated tests can be used all the time, not just during the system verification phase. The tests can be executed with each nightly build during the development phase. This process has the benefit of finding the bugs within 24 hours of the offending check-in, making it much easier to isolate the cause and get a fix in place. Without this automation, many of these errors would be caught during the regression test, and slowing testing down.

Extending this concept even further is the idea of Continuous Integration (CI), where the automated tests are executed with each check-in. In CI systems, the code check-in typically triggers a build and executing a set of tests. Typically, these tests are the unit tests, written by the developers. Unit tests are ideal for CI, as they are usually very fast to execute, and don't require setup if the environment is mocked.

Using automated tests in this fashion requires the tests to be very durable. If the tests have a high false alarm rate, the effort to triage the results will soon overwhelm any benefit from executing the tests this often. Care needs to be taken to eliminate false alarms and contain the test suite to those that run reliably and fast enough.

It's also valuable to extend the build verification process, to before the check-in. At Intuit, we developed a capability for developers to execute the build verification tests prior to check-in. The developer can upload his/her private build to the test automation system, and execute the build verification, or any tests, prior to checking the code into source control. This gives the developer confidence that the change will not break the build, and not impact others.

Tactic: Handle Defects Faster

Complexity of these systems and the fact that humans are involved virtually guarantees that there will be bugs to find and fix during the test cycles. Here are a couple of practices that we developed to minimize the impact of these bugs.

Eliminate false alarms

One productive practice was holding "bug huddles". Instead of first writing a bug in the tracking system, the testers would have a short huddle with the team lead and developer, demonstrating the bug and having an actual talk. This eliminated many false alarms, improved the fix velocity, and the fix accuracy.

One metric that we tracked was the number of "false alarms", or reported bugs that turned out to be not bugs. These were filed as "works as designed", "duplicate", or "cannot duplicate". For the first few years, when we had the leisurely 12 weeks to verify the software, our false alarm rate reached up to 50% for some releases. Some focus on our bug handling process brought the false alarm rate down to 20%.

The first project where we started using the "bug huddles", we had 27 reported bugs, from which 26 were true bugs, and 1 false alarm. The bugs were fixed faster, as the developer and tester had a much richer communication than the bug description field in the tracking system.

Meet daily to expose blockers

In the old time, we had weekly project reviews where a lot of the issues that blocked test would be hashed out. Waiting up to a week to kick-start blocking bugs was too long. In the test faster world, we had to move to a daily focus. We held daily meetings where blocking issues were raised by the team and someone assigned to solve the blockers.

To make the daily process work, we had to have real-time access to data. We had to invest in a test management system, instead of relying on spreadsheets to track tests. The test management system gave everyone a real-time view of the blocked test cases, with minimal effort to collate the results. When using spreadsheets, it took 2-3 hours each week to compile the test reports.

For this project, we ended up creating our own test management system, called creatively the Test Case Management System (TCMS). The aspects of this tool which helped our productivity was easy importing and exporting of test cases and results to Excel (via a comma delimited file) and web access so that anyone in the world-wide testing team could enter results directly, and a single database to allow real-time access to results without a data-synchronization step.

We had attempted to run this overall project using the Scrum methodology. For various reasons, Scrum did not work for this organization, but the daily meeting was valuable.

Tactic: More People

We were using an outsourced testing partner for this project, to help supplement our staff, and reduce average costs. Being the manager of the test effort, and when pressed to reduce our duration, a question the senior leaders would ask “how many more contractors do you need to pull this in?” They felt like they were helping, but adding people to a project in the short term can be more of a hindrance than a help. Adding people to projects has benefits and drawbacks in both the short term and longer term.

The model described above has a very simplistic component when it comes to labor. It just assumes that adding any person increases the denominator, thus reduces the duration of the testing effort proportionally. That is obviously simplistic.

In the short term, adding people to a project can make an already late project even later. The new people will need to know where they can help, which will add to the test manager’s workload coordinating the new tasks, and adjusting the priorities of the existing team, who are already pretty busy. The new people will likely need to be trained on how to access the test environments, where to find test cases, how to report defects, how to obtain licenses for test tools, and how to report progress. They will consume time and resources that were not in your original plan. The larger a team becomes, the communication paths become geometrically larger, which increases the management challenge.

That said, frequently an infusion of people in the short term can help the team in several ways. New people don’t necessarily have to be “drop in replacements” for the existing test team. They can relieve the existing team of some tasks, helping the existing team focus on testing. The next section describes some of our experiences with including new people and some of the tasks they can undertake.

In the longer term, adding people can help to a certain degree. Once new people are trained and have some experience, they can constructively add value to the project. Implementing some of the tactics described above is actually optimal with new people. For example, executing the regression tests earlier is better accomplished with an addition of new people on the team, since new feature test generation and old feature test execution are done in parallel, instead of serial.

Another area to add people for the longer term is in automation. The existing test team may not have the skill set to be completely effective in test automation. People with a software engineering skill set should be added. Even if the existing team has the skills, adding people to take over the manual testing is very useful.

Keeping a larger team for the long term initially sounds promising, but remember that small teams are more effective than large teams. Also, a larger team increases the cost structure of the quality team.

What Can More People Do for You?

Sometimes, the gift of new people for a testing project comes in the short term. For instance, in trying to preserve the original release date for a project that is late. In this project, we frequently received the gift of people unskilled in the art of testing. This is when the team usually pushes back with statements like “we can’t bring the new people up to speed fast enough”, “training the new people will take us away from doing our jobs”, and my favorite, “have you read the Mythical Man Month?”

However, there are sources of additional people that can be put to good use in helping pull testing in. Many of these people already have product knowledge and can add value right away, without disrupting the existing team. Likely, this will end up being a long term positive, relationships are strengthened and knowledge shared. Here are a few examples:

- **Technical writers.** These folks operate the software just as much as testers and need to understand it well enough to explain how it works. Pulling the writers into the testing may impact the development of help content & product documentation, but when push comes to shove, often the documentation can be updated after the software has stabilized and been delivered.
- **Technical Support Engineers.** Enlisting the support team is a win-win. The support team knows the product, and can think like a customer. They benefit by getting hands on training for the product they will be supporting soon.
- **Product Managers.** They specified the requirements, they know how the system is supposed to work.
- **Developers.** Its good to walk a mile in other people’s shoes. You may get some pushback from using developers to run tests because of “independence”. If so, you can mix up the testing responsibilities, a developer can test features created by other developers, or execute regression tests. I don’t buy the arguments about developers needing to be independent because they don’t want to look bad, but a very real issue is “author blindness” where the creator fails to see flaws in his/her creation.
- **Customers.** The ultimate judge of correctness. Many customers are willing and happy to help, because in the end the software is built to benefit the customers. When inviting customers in to run tests, try to avoid it becoming a “dog and pony” show, run by marketing. The most valuable is gained in hands on interactions between customers and the product, and the interactions between customers and the test staff.
- **Others in the organization.** A few times we were offered time from engineers in other areas of the company, who didn’t have too much knowledge of our domain. Also, managers like to help and pitch in. These helpers can positively help in areas that should not be too much of a distraction for the existing test team. They can help verify bug fixes, where the bug description is pretty straight forward and the bug simple to verify. They can also help review and test the user documentation and training materials, which was created to help people not familiar with the product to learn.

Summary

This was an interesting challenge, to reduce the average test duration from 12 weeks down to 4 weeks, with a product that was growing in functionality and complexity with each release. As this paper shows, no single tactic “fixed” the problem, but instead the big problem was broken down into smaller challenges and each smaller challenge was chipped away with a larger variety of tactics. We succeeded in making this change, but it was not an overnight success. This change journey happened over a 2 year period.

References & Footnotes

Pairwise Testing

Czerwonka, Jacek, *Pairwise Testing in the Real World, Practical Extensions to Test Case Generators*. Proceedings of the Pacific Northwest Software Quality Conference, 2006, <http://www.pairwise.org/docs/pnsqc2006/PNSQC%20140%20-%20Jacek%20Czerwonka%20-%20Pairwise%20Testing%20-%20BW.pdf>

Test Automation

Fitch, Todd and Ruberto, John, *Building for a Better Automation Future: One Company's Journey*, Proceedings of the Pacific Northwest Software Quality Conference, 2007, <http://www.pnsqc.org/proceedings/pnsqc2007.pdf>

Dustin, Elfriede, Rashka, Jeff, and Paul, John, *Automated Software Testing – Introduction, Management, and Performance*, Addison-Wesley Professional, 1999

Kaner, Cem, Bach, James, and Pettichord, Bret, *Lessons Learned in Software Testing – A Context-Driven Approach*, Wiley, 2001

Fewster, Mark and Graham, Dorothy, *Software Test Automation – Effective use of Test Execution Tools*, Addison-Wesley Professional, 1999

Mosley, Daniel, and Posey, Bruce, *Just Enough Software Test Automation*, Prentice Hall PTR, 2002

Exploratory Testing

Bach, James, *Exploratory Testing Explained*, April 16, 2003, <http://www.satisfice.com/articles/et-article.pdf>

Risk Based Testing

Bach, James and Kaner, Cem, *Exploratory & Risk Based Testing*, 2004, <http://www.testingeducation.org/a/nature.pdf>

Besson, Stephane, *A Strategy for Risk-Based Testing*, <http://www.stickyminds.com>

Software Reviews

Fagan, Michael, *Design and Code Inspections to Reduce Errors in Program Development*, IBM Systems Journal, 1976

Build Robust Test Automation Solutions for Web Applications

Wei Liu
Wei.Liu@erac.com

And

Dawn Wilkins
dwilkins@cs.olemiss.edu

Abstract

Test automation scripts for web applications are often fragile and lead to a high rate of false positive errors, resulting in a large amount of analysis time and maintenance effort. This paper shares our experience in improving test automation scripts for web applications and documents successful lowering of false positive (Type I) error rate which consequently reduced analysis time and maintenance effort. Causes that make test automation scripts fragile are analyzed and solutions are discussed. Notable findings include great benefit to test automation by following simple rules upstream in the software development life cycle, such as the design phase and the development phase.

Biography

Wei Liu is a senior test engineer at Enterprise Holdings, Inc. He has taken the lead role in building Enterprise's new test automation solution since he joined Enterprise two years ago. Wei gained tremendous experience in software development lifecycle and development methodology from his work as senior healthcare information system analyst at Barnes-Jewish Hospital and senior software engineer at EcommLink Corporation. He has a master's degree in computer science and is on track to obtain his doctoral degree in computer science in 2010.

Dawn Wilkins is an associate professor in the Department of Computer and Information Science at the University of Mississippi. She completed her Ph.D. in the Department of Computer Science at Vanderbilt University. Her current research interests are in the area of Bioinformatics, Machine Learning and other more conventional computer science areas.

1. Background

Ideally, any failure of an automated test script would indicate defects in the application under test. However, in reality, automated test scripts are prone to give false positive errors; that is, scripts fail through no fault of the application under test (Fewster 1999, Mosley etc. 2002). Automated test scripts for web applications are especially fragile due to the frequent changes that web applications usually undergo.

False positive errors, also called Type I errors, are costly, resulting in a large amount of analysis time and maintenance effort. Reducing false positive errors is critical in building robust automated test solutions.

Another type of error, the false negative error, is common to automated test scripts too. False negative errors, also called Type II errors, occur when an automated test script fails to find existing defects in the application under test. Reducing false negative errors is of great importance to test automation, but this paper will only focus on false positive errors.

It is worth noting that change requirements should be carefully studied and automated test scripts should be updated accordingly before regression testing of a new release. Errors that are raised by failing to properly update test scripts are not the false positive errors discussed here. The false positive errors discussed here are errors that are caused by unexpected events and cannot be prevented by studying change requirements and updating scripts beforehand.

This paper shares our experiences gained from efforts to improve automated test scripts. Though the applications that we have worked on are web-based, some of the experiences could be applied to automated test scripts in other domains too.

2. Methodology

About 200 false positive errors occurred in the past year and their respective analyses were reexamined. The errors were grouped based on their causes. Then solutions were proposed and applied for every group. The improved scripts were used in later regression tests and false positive errors were monitored and recorded. The frequency of false positive errors for the improved scripts was compared with that of original scripts.

3. Findings and Solutions

3.1 Unrecognizable Components

The most common cause that leads to automated script failures is unrecognizable components due to change of applications-under-test. In automated test scripts, webpage components are identified by combinations of their properties, such as Class, Class Index, Tab Index, Text, Value, Coordinate, ID, Name, etc. If these properties are changed, the components may no longer be recognized.

New features and hot-fixes are constantly added to new releases. GUIs are often re-designed and the look of web pages changes all the time. Not all changes can be predicted and how they affect automated test scripts cannot be determined beforehand. For instance, a developer changes the Name of a component because he thinks the Name sounds silly. Another example is adding a new component to a web page, which automatically changes the tab index property of many components on that page.

To make automated test scripts more robust in the face of changes of applications under test, unreliable properties such as class index, tab sequence index, and coordinates should not be used to identify webpage components. These unreliable properties can easily be changed without notice and it is impossible to track them in the change requests. Therefore, there is no way for automated test script maintainers to discover these changes and update scripts beforehand. These changes are not found until scripts have failed and have been analyzed. Instead, more reliable properties such as ID and Name should be used.

Moreover, the minimal set of properties should be used to identify webpage components. For example, if a component can be identified by its ID or by a combination of its class and text properties, ID should be used. This practice exposes scripts to fewer properties and lowers the chance of false positive errors.

Though sticking to the minimal set of reliable properties can make automated test scripts much more robust, false positive errors caused by unrecognizable components can be further eliminated by following the simple rules listed below at the design and development phases:

- a. Give each webpage component a unique ID or Name, if possible.
- b. Do not change a component's ID or Name unless it is necessary.
- c. Take a screen snapshot or keep a screen design for each screen.
- d. Annotate the screen snapshots or screen designs with component IDs or Names.
- e. Share this document with the Test Automation Team.
- f. Update this document after changes and notify the Test Automation Team about changed IDs or Names.

To successfully enforce the above rules in development phase good communication and coordination between test automation team and development team are necessary. Since these simple rules do not increase the work load of developers in any significant way, they should not receive much resistance from the development team.

3.2 Test Environment Change

Another source of false positive errors is test environment changes, such as operating system updates, web browser updates and automation tool updates. These changes can affect the behavior of the automated test scripts and thus introduce false positive errors. Problems caused by test environment changes can often be detected by a smoke test with a small number of automated scripts. If problems are found during the smoke test, they need to be addressed before the next regression test starts. Usually, this kind of problems can be fixed by software upgrade, reinstallation, or rollback.

3.3 Customizable Contents

Many web applications contain customizable content. Changes of the customizable content often break automated test scripts and raise false positive errors. This situation gets worse when the customized content can be changed by multiple teams, which is often the case in QA environments. In fact, this is a major source of false positive errors of test automation in the web domain. Fortunately, there are several remedies to this problem.

If an agreement that requires customizable content be restored to their default state after they are changed for whatever purpose can be reached with all the other parties involved, such as the manual test team and the development team, it can eliminate most of the false positive errors originated from this cause. However, it is difficult to enforce this agreement in reality.

An alternative is to keep a copy of clean data and use it to restore the application QA database to the “correct” state before each regression test. This solution guarantees the customizable content will be restored to the state that the automated test scripts expect. However, it needs cooperation with the DBA team and may require a lot more time and effort in the case when the database schema has changed since the last release because the saved data can no longer be simply restored.

Usually, a customizable web application has a corresponding tool, which is often web-based as well, to update its customizable content. Automated test scripts for that tool can be used to restore the customizable contents to the “correct” state before each regression test. This solution gives the automation team full control of the data and is immune to database changes. However, the automated test scripts for that tool need to be maintained.

The decision on which solution to use should be made case by case after evaluating all the pros and cons.

3.4 Exception Handling

Lack of flexibility to accommodate all possible situations the application-under-test may undergo is another source of false positive errors. Web applications may encounter many unexpected events, such as a slow network, occasional server error, and so on. Therefore, error checking and exception handling are necessary to deal with these unusual situations.

3.5 Automation Overuse

Test automation is a complicated process. The applications-under-test can be complicated, dynamic and fast evolving. Therefore, it is crucial to know what to automate and what not to automate. A common mistake is to overuse automation, which leads to enormous maintenance work and accordingly high false positive error rate.

Some tests are not good candidates for automation by nature. They either require relatively too much effort to automate or too much time to maintain while they can be easily tested manually. A typical example of this kind of test is cosmetic issues such as page layout, font size and style, and so on, which are often blamed for test failures. They are more suitable for manual test and are usually not worth the effort to automate. Therefore, it is a very delicate job to find a balance between manual testing and automated testing. Failing to do this will result in a large number of false positive errors because maintenance cannot keep up with application changes.

4. Results

The scripts which were improved in accordance with the above discussions on average have at least 80% fewer false positive errors than the original scripts in regression tests.

5. Conclusion

False positive errors are common in test automation. This paper summarizes our experience in writing more robust automated test scripts for web applications. The validity of the findings and effectiveness of the solutions discussed in this paper have been confirmed by test results.

6. References

- [1] Fewster, M., *Software Test Automation*, Dorothy Graham, Addison-Wesley Professional, ISBN 0201331403, 1999
- [2] Mosley, D. & Posey, B., *Just Enough Software Test Automation*, Prentice Hall PTR, ISBN 0130084689, 2002

Too much automation or not enough? When to automate testing.

Keith Stobie

Keith.Stobie@microsoft.com

Abstract

Fundamentally test automation is about Return On Investment (ROI). Do we get better quality for less money by automating or not automating? The obvious and famous consultant answer is “it depends”. This paper explores those factors that influence when to choose automation and when to shun it.

Three major factors you must consider:

- 1) Rate of change of what you are testing. The less stable, the more automation maintenance costs.
- 2) Frequency of test execution. How important is each test result and how expensive to get it?
- 3) Usefulness of automation. Do automated tests have continuing value to either find bugs or to prove important aspects about your software, like scenarios?

Biography

Keith Stobie is a Test Architect for Protocol Engineering team at Microsoft working on model based testing (MBT) including test framework, harnessing, and model patterns. He also plans, designs, and reviews software architecture and tests.

Previously Keith worked in Microsoft's Windows Live Search live.com and XML Web Services group. 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

Books on Test Automation (Dustin99, Mosely02) usually spend a few pages on the question of whether to automate. Fundamentally test automation is about Return On Investment (ROI). Do we get better quality for less money by automating or not automating? The obvious and famous consultant answer is “it depends”.

Three primary drivers are cost to create, run, and maintain, but there are many other differences between automated and manual tests. Further, it is not an all or nothing choice. Automation can be done for some test cases and not others. It can even be done for parts of a test case and not others.

More subtle is the realization that an automated test is rarely the same as a manual test. A manual test is performed by a human with human capabilities of observation, and ideally given the charter to improvise to enhance its value. An automated test is usually limited to what its creators thought about and coded. Other characteristics of manual vs. automated tests, such as metrics, are not considered in this paper. Suffice it to say that you get different data from these two types of tests with varying levels of accuracy. If the metrics and accuracy are of primary concern, and not cost, then these other factors must be considered.

2 Why Repeat

Repetition is the first key question about when to automate testing, but not the only question. Sheer repetition is typically sufficient to get ROI for automation. However, you must understand why you need the repetition to justify the investment. Repetition for no reason is just a waste of money.

If there were no defects to be found, there would be no testing needed. Testing provides information to give confidence, but if you knew there were no defects, that information would be of little value. I don't know of any large, modern, complex software systems where anybody knows there are no defects. When doing conformance testing, the defect being checked for is lack of conformance.

The frequently stated reason for repetition is: any repetition has the possibility of a finding a defect if something has changed. That something could be the inputs, the software, or the environment. While it is true that almost anything has the possibility of finding a defect, testing is the art of choosing those activities having the greatest possibility of finding defects of the most consequence. Choosing which defects are of most consequence depends on context. A typographical error in an error message for a computer game may be of little consequence, while that same error in a log message for enterprise operations to act on could cause operators to take actions resulting in the loss of the system.

Most customers are more upset when a previously working feature fails than when a brand new feature fails. But if the brand new feature is the reason for purchase and not the old features, then new features working becomes the more critical factor. Thus the consultant's statement “it depends”. You must know what is more important for your customers.

Whether repetition helps find defects of the most consequence will significantly impact your choice to automate testing. Choosing what to repeat is critical. The cost of repetition also factors highly into the equation.

2.1 Change the input values – data driven

As an example, in designing a test case to verify the square root function for a 32 bit computer the following options are available:

If you are worried about testing the function

- just once with one input,
then manual testing will be most cost effective.

- with a few sample points only a few times, then manual testing may be most cost effective.
- with a few sample points, but many times, then automation becomes imperative.
- completely, even just once, then automation is imperative. Verifying 2^{32} inputs and their corresponding output by a human will vastly outweigh the cost of automation.

Testing a simple square root function has the cost of automating small relative to the cost of manual verification. Even testing a 64 bit computer's square root function is practical assuming you have over a million computers (say 2^{20} spread over the internet by volunteers), and each value can be tested in a nanosecond, resulting in about 2^{30} values tested each second on each computer. Then only 2^{14} seconds or about 4.5 hours are needed to verify all possible inputs and resultant outputs.

However, just changing from the simple world of integers (or floating point numbers) to strings, makes the all values testing impossible. Each Unicode character in a string is roughly one of 2^{16} characters. If I have 8 characters, then there are 2^{128} possibilities. With the same resources above, it would take 4.5 hours * 2^{64} or about 10 quintillion years. So, except in rare cases, verifying all values is not possible.

2.1.1 Value Selection

How many values to sample to give confidence and which values are most likely to find defects of consequence is the subject of many testing books and papers (e.g. Whittaker02). This paper presumes you know how many values you need to sample. A difference between automated and manual tests is the strategy for choosing values of consequence. In either case you can fix the set of values ahead of time (perhaps chosen based on boundary value analysis or other techniques), which makes it merely a question of repetition. With automated or manual tests you can provide different dynamic strategies. Automated tests may use a random or adaptive random testing strategy. Exploratory testers may use intuition and past observations.

2.2 Changing the environment – setup

Besides changing the input values, another area to vary is that of the environment. You may want to run using different versions hardware or of related software including different operating systems, languages, etc. The setup costs for these different environments will drive the choice for automation. Many companies are using Virtual Machines as a means of lowering setup cost. If tests must be repeated on different platforms, versions, languages, etc., then all of those repetitions increase the frequency of test execution. Making automated tests configurable to handle the different platforms, versions, languages, etc. may also have an added cost.

For example, one company may need to just sample a few times whether their software works when a USB device is unplugged and plugged back in. Manual verification is probably their cheapest solution. Another company needs to verify many pieces of software against many different USB devices being unplugged and plugged back in. The second company may find a \$100,000 investment in machines to mechanically plug and unplug USB ports cheaper than hiring manual testers to do the same work over many months or years.

2.3 Changing the software – regression

The third area of change, the software, depends on the nature of the change. Is the change a refactoring that is not supposed to affect the external behavior of the software? Then automated testing becomes more feasible because the expected outputs are not expected to change. Is the change a new aspect of the software that cuts across many features and changes their external behavior? Then either manual or automated testing will have to account for the new behavior. The manner in which the manual or automated testing is performed will affect the cost of accommodating the new behavior. Many books about software, testing and test automation deal with various technologies to reduce the cost of

accommodating change (e.g. Gamma94). This paper presumes you have an understanding of your costs for accommodating various types of changes.

See also the Test Maintenance section later for further discussion.

3 Latent vs. Regression defects

After a test with specific input on a specific environment, against a specific software build or version shows the software behaves as expected for that input and environment, then there is virtually no possibility that the test will later show the software does not behave as expected, unless the input, environment, or software changes.

If a defect exists in your software, but has not been discovered by your testing process, it is a latent defect. Changing the inputs or environments without changing the software may reveal where the software does not behave as expected for the current build or version. This can show latent defects.

Changing the software to a new build or version, without changing the inputs or specific environment may reveal changed outputs. If the changed outputs are expected, then tests require corresponding changes. If the changed outputs are not expected, this is called a regression. A regression is an unexpected change in output due to a new version of software. The unexpected change in output is that it doesn't match the output of the previous version.

From a zero based budgeting perspective, you have at every point in time the following choice: look for more regression defects or more latent defects. At any time your software is at a state where you've identified (and hopefully usually removed) known defects found by your testing process. Specifically you can repeat an existing test on changed software, or you can run a new test on the software (changed or unchanged). If your software hasn't changed and you have run all the regression tests you desire and you have remaining time and budget, then latent defect testing is the obvious way to go. However, if you don't have time and budget to run all of the regression tests you desire (almost always), then you have the hard choice of whether to run a regression test or a new test.

Many test managers focus almost exclusively on regression testing just before shipping. If regression testing finds the most defects of greatest consequence, then this is a wise choice. As stated earlier, a business context determines whether finding regression defects is more important than finding latent defects.

The context in which your software operates is critical. Safety critical software that affects human life costs more because the testing must be more thorough, repeatable, and justified. Similarly software that has significant financial impact is usually funded for more testing. On the other hand, a small custom application for a single department of 20 users doesn't have the same funding for testing, nor the same need for automation.

Running a test the first time, whether automated or manual, is about finding latent defects (existing defects missed earlier). Running the same test again with only new software is usually about regression defects (new defects unexpectedly introduced). Over time concern frequently shifts from latent (incomplete product testing) to regression (changed product) defects, and thus manual testing frequently shifts to automated testing.

Numerous studies and papers have shown that detecting and removing defects early saves money. Thus detecting latent defects and regression defects early saves money. Typically manual tests are most cost effective for finding latent defects early. If the manual test finds the behavior expected, it is not repeated. Typically automated tests are most cost effective for finding regression defects early as they can more economically be repeated after frequent and small changes.

Note that even after you have automated tests, regression testing is not free. If any unexpected outputs occur they must be analyzed to determine if the test software or the software under test (SUT) is at fault.

Both being software, they are both likely to have defects resulting in the unexpected result being observed. See Examples of Automated and Manual analysis for further discussion.

3.1 Capture/Replay

A frequent, but sometimes dangerous way to change from manual tests looking for latent defects into automated tests looking for regressions is to record the manual test in some fashion and then be able to use that recording as a way to repeat or replay the test. This can be done at any level, code, User Interface (UI), protocol on the wire communication, etc. This major issue is the assumption that a manual test should be transformed into an automated test. Many latent defect finding tests aren't worth repeating. Their value in finding a latent defect was perhaps marginal and repeating them provides even less incremental value.

Given that automating a test is more expensive than running a test manually, which regression tests to automate should not default to all of the attempts made at finding latent defects.

4 Manual vs. Automated tests

Human versus computer execution is emphasized in lesson #108 in Kaner01 "Don't equate manual testing to automated testing". Manual tests can also be frequently described less exactly than automated tests. To have an automated test, it must ultimately result in code executable by a computer, and most instructions to computers today are fairly exacting with formal syntax and rules, whether a scripting language such as Ruby or Powershell, or a compiled language such as Java or C#. Automated tests using keyword or action word automation elevates the abstraction language, but is still ultimately turned into a detailed sequence by software.

4.1 Exploratory Tests

Manual tests may have only broad charters that employ exploratory techniques of concurrent iterative plan, design, execution, and evaluation of test ideas. Exploratory testing is effectively only possible today using humans doing some form of manual testing. Some types of testing are best done using exploratory testing. Usability testing in terms of how humans perceive an interface generally requires humans. You can perhaps use automated tests to verify user interface guidelines, but not necessarily usability.

4.2 Scripted Tests

The other form of manual testing can be called scripted testing that describes step by step procedures. Still the level of detail of the steps can vary widely. It might be high level and loose, for example:

or `Login`
`Enter username and password`

Or low level and very tight such as

`Click mouse in username field to place cursor in username field. Type "testuser".`
`Click mouse in password field to place cursor in password field. Type "Pa$$w0rd"`
`Click on login button`

The second script is approaching the level of detail eventually required for an automated test (although an automated test might also have high level methods like

```
LoginEnteringUsernameAndPassword(string username, string password)
```

This results in execution similar to the detailed script.

The first script allows the human, manual tester, a great degree of freedom in how to approach the task.

They can choose the method of entry: keyboard short cuts? Tabbing to fields? Using the mouse?

They can choose the data to enter: if multiple usernames with passwords are available they can use different ones at different times.

When scripted tests use high level loose steps, they are cheaper to create and cheaper to maintain than low level tight steps. However the initial execution cost of high level loose steps can be higher since it requires more domain knowledge. Assuming the same human (or set of humans) repeats the tests within a few days or perhaps weeks' time frame, then repeating a high level or low level script is roughly the same cost. The cost for high level running goes up when domain knowledge is lost due to change in personnel or infrequency of repetition.

4.3 Manually written tests

The cost of automating tests can vary widely also depending on tools and technology available and implementer knowledge. Automated tests are just a form of software and studies have already shown a 10 to 1 factor in skill level of programmers. That is, a very skilled programmer can, in one hour, create as much functionality as a poorly skilled programmer does in ten hours.

Most automated tests today are created by hand coding. They may be specified by a subject matter expert using keyword driven testing and automated by a specialty test interface team. They may be written by software engineers with only a cursory knowledge of the domain. In most cases defining and automating a test (or set of tests) and executing it once is more expensive than defining and manually executing a test (or set of tests) once for the same functionality.

4.4 Automatically generated automated tests

Instead of creating each test by hand, tests can be automatically generated. Simple methods of automatic generation include data driven tests. More advanced test generators are frequently based on formal models such as grammar models or behavioral models (such as state diagrams).

4.5 Test Repeatability

Using either detailed scripted or automated tests we can get highly repeatable tests. Automated tests are typically more repeatable as most humans have a difficult time exactly repeating detailed instructions (for an exception see Saran08).

Test automation is ideal for showing that for a few data points the software behaves the same as before. This can come either from manually written automated unit tests or from integration tests to show a software build appears as stable as a previous build. These are frequently called build verification tests.

4.5.1 Unit Test Repeatability Essential

In the Agile community I heard stated that all code written without automated unit tests is legacy code. Legacy code is the old, decrepit code no one wants to touch for fear of breaking. If you don't have unit tests, the risk of changing code and inadvertently introducing a defect increases significantly. When deliberately introducing a breaking change, the chance of side-effects is even higher. Unit tests make code changes safer. Automated unit tests usually make code safer than manual unit tests as they are more repeatable (does it appear to "work" as well as it used to) and are normally run more often.

Manual testing is rarely useful for repeatable unit testing since it is very low cost to transform most tests at the unit level from manual tests to automated tests, and running them frequently provides the early regression capability needed. The cost of manual execution of unit tests usually causes them to be executed less frequently and degrades their ability to act as early regression tests as further described in Marick00.

4.6 Test Maintenance

Don't underestimate the maintenance problem. Because of maintenance, often tests are picked because they are easy to maintain for automation, not because they will necessarily result in the greatest increase in product quality. For example, customers might use a mouse most when interacting with a GUI and the

customers might find numerous issues because it was easier for the test team to automate the GUI with keyboard short cuts rarely used by the customer base.

Maintenance costs for automated tests, like software, depend on the original design and architecture. Software design patterns and many other software techniques have adaptability as one of their concerns. If the software was not written to be adaptable, but must change, the cost of test maintenance can be quite high. If the software is written to be adaptable, the cost of change can be quite low.

I have seen the same for automated test suites. Badly designed and architected GUI automated tests are very brittle. I've seen a test team say it would take 2 weeks to update their tests for a change to the product software that took less than an hour to design, create unit tests, code the change, and verify it with the unit tests. I've also seen a test team indicate it would take 15 minutes for them to update their test software due to a change that took 2 days to implement in the product software.

If you automate UI testing early and the UI changes every day in multiple ways, the best case is you keep up with the UI in your automation. The worst case is you quickly fall hopelessly behind.

5 Return On Investment

Testing is a business activity with a cost. To calculate the ROI for test automation, you must be able to measure the quality of the software being tested. This is a topic unto itself addressed in other books and papers. Secondly, you must measure the cost to get that quality. As Hoffman points out (Hoffman99) there are many intangible costs that are difficult to realistically assess. The goal with test automation is increased quality for the same cost or reduced cost for the same quality.

Using your resources wisely to most effectively cover the risks requires:

- understanding the product and release goals
- dealing with common issues like known errors
- anticipating likely errors
- choosing the right evaluation methods

In doing any ROI calculation, the high degree in human variance must be allowed for. That is, it may not be determinable in many cases whether manual testing or automated will work better since other factors such as personnel skill and technology known could dominate the cost.

Assuming your organization has determined the level of script detail it wants and alternatively what software technology it will use with which skill level of staff, then you can begin to do some comparisons. Frequency of test execution *alone* is not a good measure, as given by lesson #109 in Kaner01, "Don't estimate the value of a test in terms of how often you run it."

Related to the ROI is the opportunity cost of manual testing versus automation. Initially, for most types of testing, you can test more once manually than via automation. But the more you retest without having to do significant maintenance or alterations, the more attractive automation becomes. Marick expresses this (Marick98) as:

If I automate this test, what manual tests will I lose? How many bugs might I lose with them?
What will be their severity?

Three major factors you must consider:

- 1) Rate of change of what you are testing. The less stable the SUT, the more automation maintenance costs over loose scripting or exploratory testing.
- 2) Frequency of test execution. How important is each test result and how expensive to get it? Running all tests all the time is not economically feasible. It might be very inexpensive to run automated tests, but unless they reveal defects of consequence, they may curtail your investment in more expensive manual tests and more rarely run tests that might reveal more defects of consequence.

- 3) Usefulness of automation. Do automated tests have continuing value to either find latent bugs or to show regressions? At the system level, automated tests are usually targeted around regression testing scenarios and non-functional qualities such as performance or security.

So far the rules of thumb for when to automate and when not to automate boil down to:

- **Unit tests** are typically **automated** – their automation cost is typically a low multiple of their manual test cost, they are run frequently while a product is evolving, their maintenance cost is usually not more than the cost of changing the code itself.
- **Build verification tests** are typically **automated** – their automation cost is typically outweighed by the number of times they will be repeated, the consistency of information desired (same exact values), and moderate maintenance costs.
- **Usability tests** typically use **manual exploration** – they are nearly impossible to automate, not typically affected by every software change and thus have no need to repeat as frequently, and have potentially very high maintenance costs.

5.1 Application Programming Interface (API) Testing

Most API level tests are typically automated, since code must be written to invoke the APIs. You can create interactive shells that allow calling APIs from an interactive scripting-like environment, but they sometimes limit you versus the native implementation. You can also automatically record the exploration and nearly automatically turn it into an automated test script. Capture/Replay for APIs is more reliable than for User Interfaces (UIs, such as command line or graphical). Exploring APIs in a non-automated fashion is a way to verify usability of an API, but so is building software similar to the anticipated usage of the APIs using scenarios. Finally, APIs easily lend themselves to many of the simpler analysis techniques so that creating high coverage automated tests should not be much more expensive than manually testing APIs.

In any testing, you must understand the nature of what you are testing and how deterministic it is. Non-deterministic results may make automatic testing difficult regardless of API or UI. Non-deterministic outputs from machine learning and artificial intelligence programs are particularly difficult. This includes grading results from a Search or a complex distributed system.

A simple example I experienced was a set of SQL tests. The SQL Language does not guarantee that the results of 3 inserts into a database will return those 3 items in the order inserted. However, testers naively observed and assumed that. When the underlying SQL engine was changed into a parallelized version, most of the tests broke because they made assumptions beyond those allowed by the APIs. Writing tests to not assume more than the API can be very difficult at times. Note, that it doesn't get any easier for manual testing humans either as now noticing whether 1 record out of a 1000 is missing is quite error prone. Most humans are not good at detail as evidenced by typographical errors overlooked by humans every day.

5.2 Performance and Stress testing

If you want to monitor performance over time to check for degradation or improvement, then you want repeatable performance tests. The more often you want to repeat them, then the more likely automated tests will be applicable. If you are verifying small, atomic operations like a single API call or Database retrieval, then automation becomes critical because the time scale is too small. Any performance testing that takes under a few seconds would be unreliably performed by humans.

Load and stress tests generally use automation to create the load and stress. However, I have seen reports of large manual load or stress tests. This is especially true when testing a large, worldwide distributed, production system. A typical compromise is to combine automated testing for generating the load and stress with manual exploration concurrently looking for anomalous behavior the automation can easily miss.

5.3 Software Version stages

Beyond the types listed above, there are still numerous areas for consideration. Some questions to consider:

- Should integration and system tests be automated?
- Where are your company and its product in its lifecycle?

The ROI for automation frequently occurs after the first release. If just getting a quality first release is the biggest concern, as for example in a startup, then defect finding at the system level via manual testing may be of more benefit than automation to save future costs. When a company has an expectation to maintain the software for 10 years and even if developing the first version of a product, then future cost reduction via more spending now on automation of system testing may be of most benefit. A company may choose to ship fewer features in order to fully automate the ones shipped. Another frequent approach is loosely the following for version V of a product:

- Assume mostly automated tests for the previous version, say V-1.
- As version V is developed, new features are manually tested, old features are automatically regression tested using the V-1 test cases.
- When version V is shipped, as planning is beginning for V+1, the V test cases are automated.

There are many dangers to the above approach:

- 1) If planning for V+1 occurs during the end of the version V cycle, then there is never time for test automation.
- 2) If test is automating version V tests at the start of the V+1 cycle, then test may not have time to participate in the planning (and Marick questions whether they should, Marick99).

So lifetime of the product will affect the anticipated number of reruns of the test cases, but other factors can still affect when they should be automated. Perhaps you don't automate the test cases until after the product being tested proves to be profitable.

6 Partial automation – S.E.A.R.C.H.

Automation is not a binary question. Good tests typically have six parts as described with the acronym SEARCH in Bergman92. SEARCH stands for **Setup**, **Execution**, **Analysis**, **Reporting**, **Cleanup**, and **Help**.

- Setup** provides the assumed environment (either by verifying it exists or creating the environment).
- Execution** of the test is running the software. Code coverage tools measure what was executed, not what was verified.
- Analysis** is the verification. It may be asserts in the test code as it ran or it can be post processing of whether the software did what it was supposed to do and didn't have any unintended side effects on the system as a whole.
- Reporting** the results of the analysis can include roll ups and statistics.
- Cleanup** is the final phase that returns the software to a known state so that the next test can proceed.
- Help** is the associated documentation for the tests: how they were designed to operate and how to keep them runnable and maintainable. This is particularly true regarding which tests can be run in parallel and which must be run separately.

Each of these can be done in an automated or manual fashion for justifiable reasons. I've seen testers automate their execution time down to 10 minutes while repeatedly doing manual setups of one hour. They should have focused their attention on Setup.

6.1 Examples of Automated and Manual analysis

When tests are repeated in many environments, then analysis of failures (not their detection), may become a predominant cost, so automated failure analysis may be the most useful automation.

Automated Failure Analysis is a means of determining if a specific observed failure has previously been diagnosed. See Trivison08 and Staenff07 whose subtitle: "Why Weak Automation Is Worse Than No Automation" really strikes home.

Another example is verifying UI output. Frequently minor variances in UI can be critical or inconsequential. Computers have a difficult time distinguishing. Computers can easily distinguish if something is different, but not the significance of the difference. So one example of automation is to automatically record differences (analysis) and later provide humans the ability to manually evaluate their significance (reporting).

7 Conclusion

How are you repeating your test? Manually via humans or automated via software? Software test automation is merely a method of repetition. When to automate software testing is dependent on many factors mostly around repetition.

When are you repeating your tests? How often? Every input value or combination of inputs? Every day or once a year? The more you repeat the more likely automation makes sense.

Why are you repeating your tests? Change to the inputs, environment, or software? Changing the inputs requires an understanding of what the expected outputs will be. For UI testing, a human can sometimes more quickly and easily judge if the expected output looks right. For changing environment and software you are usually expecting the same output, and automated tests are typically more reliable at detecting changes.

What are you repeating about your tests? What are you varying? Repeating the same inputs makes automation more likely. Varying an environment may be expensive and difficult and thus only manually done. Repeatedly using the same SUT means you are looking for latent defects, not regressions, and exploratory testing may be most valuable.

8 References

Bergman M., and K. Stobie, 1992 *How to Automate Testing-the Big Picture*, Quality Week 1992
http://keithstobie.net/Documents/TestAuto_The_BigPic.pdf

Dustin, E., et al 1999, *Automated Software Testing: Introduction, Management, and Performance*, Addison-Wesley Professional 1999. ISBN: 978-0201432879

Gamma, E., et al 1994, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994 ISBN: 978-0201633610

Hoffman, D. 1999, *Cost Benefits Analysis of Test Automation*, Proceedings of the Software Testing Analysis and Review Conference (STAR East). Orlando, Florida. May 1999
<http://www.softwarequalitymethods.com/Papers/Star99%20model%20Paper.pdf>

Kaner, C., J. Bach and B. Pettichord. 2001, *Lessons Learned in Software Testing*, Wiley 2001

Marick, B. 2000, *Testing For Programmers*, (pages 43-47)
<http://www.exampler.com/testing-com/writings/half-day-programmer.pdf>

Marick, B. 1999, *Maybe Testers Shouldn't Be Involved Quite So Early*
<http://www.exampler.com/testing-com/writings/testers-upstream.pdf>

Marick, B. 1998, *When Should a Test Be Automated?*, Quality Week 1998
<http://www.exampler.com/testing-com/writings/automate.pdf>

Mosley, D. and B. Posey 2002, *Just Enough Software Test Automation*, Prentice Hall PTR, 2002.
ISBN: 978-0130084682

Saran, c 2008, *Autistic people prove valuable in software testing*, ComputerWorld, 15 Feb 2009,
<http://www.computerweekly.com/Articles/2008/02/15/229432/autistic-people-prove-valuable-in-software-testing.htm>

Staneff, G., 2007, *Test Logging and Automated Failure Analysis*, SASQAG September 2007.
http://www.sasqag.org/pastmeetings/9-2007/SASQAG_9-20-2007.ppt

Travison, D. and Staneff, G. 2008, *Test Instrumentation and Pattern Matching for Automatic Failure Identification*, 1st International Conference on Software Testing, Verification, and Validation, 2008
ISBN: 978-0-7695-3127-4

Whittaker, J. 2002, *How to Break Software: A Practical Guide to Testing*, Addison Wesley 2002
ISBN: 978-0201796193

Some Observations on the Transition to Automated Testing

Robert A Zakes

Requirements and Testing Manager
Information Systems Division
Oregon Secretary of State
robert.a.zakes@state.or.us

Abstract

We started on our test automation effort about two years ago when we realized testing was slowing our application development and our agency faced monumental development projects.

After some research we decided to use Selenium as our test automation tool and started scripting. Lisa Crispin's workshop at the 2007 Pacific Northwest Software Quality Conference (PNSQC) gave us some valuable insights and several attendees validated our choice of Selenium. We now have extensive regression scripts developed for most of our systems and run them religiously for every build.

This paper relates our observations along the way. We found test automation had many uses besides building regression scripts, and they will be described in this paper. It will also describe the process of how to benefit from the maintenance of regression scripts for a new release. In addition, this paper will examine some lessons learned, describe our focus for ad hoc testing, and present some metrics of manual vs. automated testing.

Biography

Bob Zakes is currently the Requirements and Testing Manager for the Oregon Secretary of State's office.

Bob has over 40 years experience in project management and software requirements and testing and has a BS in Engineering along with an MBA from the University of Illinois. He has worked for IBM as marketing representative, instructor, and marketing and systems engineering manager; National Retail Systems West as product manager; Hanna/Sherman International as product and engineering manager; and the State of Oregon, where he has served as project manager for several successful system implementations. He is currently responsible for testing Oregon's Central Business Registry, the Public Records System, and ORESTAR (Election Reporting).

How We Tested Prior to Automation – Manual Scripts

Our earlier testing procedure started with a test plan that outlined test cases and combined them into test scripts. Initially, these scripts served as the acceptance criteria for new applications. When the scripts could be completed without error, the applications were ready for implementation. As the applications and the build process became more complex, we started using the scripts for regression testing. They were designed to prevent the age-old software development issue of fixing one thing and breaking another.

Our scripts were written in Microsoft Word in a table format. A script document contained setup instructions, references to source documents if data was not embedded in the script, and one or more test cases. There were blanks for the tester name, test date and time, and other relevant variables.

Each test case included a test case number, a description, instructions, expected results, and a place to record actual results. The instructions described every field entry and every mouse click. The expected results described what should happen after each instruction was executed, in terms of navigation, validation, status, *etc.*

§

BERI ePay 5.6 Migration – TEST SCRIPT

#	Test Case	Setup & Instruction	Expected Results	Actual Results
6	Cardholder Name Entry Page, validate edits and navigation	<p>On Cardholder Name Entry Page</p> <ol style="list-style-type: none"> 1. Attempt to continue without entering any fields 2. Attempt to continue by entering only a space in required fields. 3. Attempt to continue by only entering a first and only a last name and only a middle initial 4. Try to enter multiple characters in the MI field 5. Selectively test to see if each required address field (marked with *) is really required. 6. Enter Cardholders Name & Address, <p>Your Name Your Address</p> <p>Click on Contact Us and Back</p> <p>Click on Privacy and Back</p> <p>Press Previous and Back</p> <p>Press Next, go to Case 7</p>	<ol style="list-style-type: none"> 1. Required fields must be entered. 2. Required fields must contain valid characters 3. The name must have both a first and last name. Middle Initial is optional 4. Middle Initial should be only one character 5. All required address fields must be entered 6. Name and Address does not change when going to another page and back. 	<ol style="list-style-type: none"> 1. Required fields ____ 2. Valid characters ____ 3. Name test ____ 4. MI is one character ____ 5. Address test ____ 6. Name and Address does not change ____

Figure 1. Sample Test Script Page.

The authoring process normally could not begin until there was a working prototype of the screen page. The script author would think through the setup and data requirements, often filling out paper forms to serve as the test data, and then enter the instructions and the expected results. Testing navigation and validation/error messages required checking the requirements and use case, then executing the function to complete the script. Once scripts were developed, they had to be tested. We tried to get someone other than the author to run the script to ensure it could be run and was not confusing.

Because this process was labor-intensive and few had the required talent or patience, scripting frequently was the bottleneck of our development cycle. A typical test case for a single web page would require 8 to 16 hours of

scripting. A typical script with 10 test cases would take 2 weeks to a month to prepare. Most applications required multiple scripts, and with only a few scripters this process could take months.

Once the scripts were complete they were run for application acceptance. After each run, completed scripts were collected, reviewed, and filed. The acceptance criterion was for all scripts to finish with no Priority A errors (no work-arounds) and minimal Priority B errors (work-arounds available, but onerous). Each rerun became more and more difficult to schedule and more and more stressful for all parties as deadlines passed.

Manual testing also was affected by many personnel issues. There is a wide variance in testers' abilities to test correctly. When non-repeatable errors occurred, there was always the question of whether the tester or the system was the cause. Scheduling issues came into play: different work schedules, illness, and vacations. Stress became a factor for user department testers when regular work was being deferred. Finally, maintaining accuracy when repeating the same script set was a continual problem.

Script maintenance also was very time-consuming because it was difficult to update instructions and expected results within a Word table. The focus was on editing, rather than testing. The revised scripts had to be printed and retested. It took considerable commitment to keep scripts up to date after an application was in production, and even more commitment to run them religiously after every build.

As the number and complexity of our applications increased, testing became the bottleneck. We were doing everything properly, but did not have the resource to keep up with the demand. Our agency's visibility did not allow reducing our commitment to quality, so we had a dilemma: how to streamline testing without sacrificing quality.

How We Test Now

We realized our problem was due neither to our knowledge nor to our commitment to quality; rather, it was due to our manual process of writing and running test scripts. We looked at how we could speed up the process. We looked at test automation but were apprehensive of the perceived cost and complexity of automated test systems. We asked our primary development contractor, Zirous, for alternatives, and they suggested Selenium as a solution. After some additional research we decided to use Selenium as our test automation tool. Some of Selenium's benefits include:

- Selenium IDE is not simply a recording tool; it is a complete Integrated Development Environment (IDE). Users may choose to record, edit and run scripts interactively.
- Selenium IDE is implemented as a Firefox extension allowing users to record and play back tests in the actual browser environment.
- Intelligent field selection uses IDs, names, or XPath as needed.
- A test debug mode includes the ability to set execution speed and set breakpoints.
- Tests can be saved as HTML, Ruby or JAVA scripts, or several other formats.

We started scripting with Selenium in May of 2007. You can download Selenium and add it to Firefox in 5 minutes and begin scripting. It is open source and has been developed and maintained by OpenQA. Within an hour you can create useful test scripts. We started using Selenium IDE as our primary scripting and testing tool and have been using it ever since.

Lisa Crispin's workshop at the 2007 PNSQC, "Getting Traction with Test Automation," gave us some valuable insight. Her workshop showed how test automation can become an integral part of agile development. Several workshop attendees had been using Selenium and related their success.

Although scripts can be created and run immediately after downloading Selenium IDE, we initially encountered several limitations. We had difficulty with scripts stopping during page loads, and the IDE would not record some Ajax elements or dynamically created links. At first we did not know how to use flow control, so our early scripts were huge due to cutting and pasting repetitive validations. Two years ago the extent of Selenium documentation was very limited. However, even our initial, primitive scripts offered such improvement that we forged ahead.

One procedure that helped our progress was to keep a detailed list of limitations and review them with peers and ask for help from the OpenQA forum. The solution often required Java Script. We used a Java developer who coded commands in Java Script for some of these situations.

Two years later, we have used automated scripts to speed the development process and have automated regression scripts for all our major Web-based applications. These scripts run for hours without stopping, and our list of limitations has diminished considerably. We will now describe some methodology and then cover observations.

Planning

Test planning is more critical when you are in an automated environment. Some of the issues we address as we plan for a new application or major upgrade include:

Script Design and Content. Our planning identifies what test cases will be tested in which scripts. One key decision is script size. We build large scripts that perform beginning-to-end testing; the tradeoffs of small modular scripts vs. end-to-end tests will be discussed below. Occasionally, it makes sense to create a separate script. For example, when the function to be tested is very complex, rather than add that complexity to every script we test it in a separate, smaller script. Another example is leftover or exclusive paths or processes that don't fit logically into any scripts; we combine these into an "everything else" script. For instance, we generally create a separate script to test side links like FAQs and Privacy.

Sufficiency. Automation makes it possible to test every permutation, but in some cases it is very easy to go off the deep end. We have a routine that checks for duplicate individuals based on name and address. There are ten variables, resulting in 3.6 million tests. This could be scripted by indexing through all the permutations, but how much is sufficient? We determined thirty tests provide adequate code coverage.

In-line vs. Subroutine. You can build subroutines to test common functions, or you can cut and paste these tests into multiple scripts. In the previous example, we built a subroutine to test for duplicate individuals and used it for the six occurrences. We could have copied and pasted it into each page to be tested, but the script would be very large and difficult to maintain. We have subroutines for testing individual and business names, addresses, telephone numbers, signature authentication, and other fields.

Need for Extensions. We try to identify the need for special tools or commands by exploring each new page with Selenium to verify we can identify all elements. We don't want to hold up a release or have to manually test some new function that we can't script.

Identify Changes That Have a Large Impact on Testing. We evaluate how new functions can be scripted as soon as possible. Sometimes minor changes can mean days of script rework. The key is to communicate to the team that the feature or change could delay the release and look for alternatives.

It is worth re-emphasizing the point: a little planning can save a lot of scripting.

Building Scripts

Open Firefox. Open Selenium. Selenium starts with Record Mode on. Sign on and open the first page. Complete each field and save. Selenium has created "type" commands for every entry field and "select" commands for every click, and we have just created a "happy path" script (one that goes through the page with minimal entry and no errors) for this page. This script now can be stored and repeated. See sample.

Sample Selenium Entry Script		
Command	Target	Value
waitForTextPresent	Authorized Representative Information	
type	indvFName	Aaron
type	indvMName	Authorized
type	indvLName	O'Representative
clickAndWait	btnContinue	

Note: The open, sign in and address entry are not shown.

Open a page and save. Normally, several validation messages for all the mandatory fields will be displayed (e.g., “First Name is required.”). Click on each validation message and select the command “verifyTextPresent” from Selenium’s popup command menu. Selenium records the validation for each error message. This process takes minutes. The result is a script for the page’s mandatory field checks. It can also be saved or combined with other scripts.

Sample Selenium Validation Script		
Command	Target	Value
waitForTextPresent	Authorized Representative Information	
clickAndWait	btnContinue	
verifyTextPresent	First Name is Required.	
verifyTextPresent	Last Name is Required.	
verifyTextPresent	Address Line 1 is Required.	
verifyTextPresent	City is Required	
verifyTextPresent	Postal Code is Required	

Note: Only the result of the first attempt to save is shown.

We started by building small test modules and linked them together with Selenium Test Runner. Test Runner permits linking several test scripts to run sequentially. With the stack of modular mini-scripts, we were isolated from the problem and would have to open the IDE, open the script with the problem, position the application to the module’s start point, and step through to find the problem. We opted to build long scripts (4,000 commands) that run for 1 to 2 hours. Running the scripts using the IDE allows us to diagnose a problem as it happens, fix the script in line, and continue. This interactive ability to debug and build scripts as you run offers a huge advantage in productivity.

Scripting During Development

We test and build scripts concurrently as the application progresses and continue to run them with each build. We review the use case and our Script Check List (see below) as we test to ensure we have tested everything for that page. As the application matures, the scripts mature and the result is a suite of regression scripts.

Scripting during development is helpful in other ways:

1. A repeatable condition can be set up rapidly for trial-and-error testing (e.g., find the best way to code a query).
2. Scripts can be run to generate an error condition and diagnostic tools can be run as the error is executed.
3. Scripts can enter field-identifying data. Field-identifying data refers to the technique of entering the field name and context into text fields, such as “Owner Four Addr 1” into the fourth owner’s first address line. This can be followed through the application to ensure the original entry is saved and redisplayed and printed correctly throughout the application and its interfaces. This technique is also useful for checking interfaces, XML, database saves, translations, etc.
4. “Happy path” scripts with switches and stops can be used for debugging. Switches can be used to set different test environments and major paths through the code, and stops can be used to stop the script at any page in the application. The script can stop several pages into an application with a repeatable entry sequence to help diagnose a problem.
5. It permits easy generation of permutations or combinations for unit testing.
6. It provides a method to create repeatable test data quickly.
7. It ensures refactored code continues to provide previous functionality.
8. It identifies inconsistent element naming conventions.

9. It helps find the cause of intermittent problems three ways. First, it allows for better detection since the code is typically executed many more times so there is a higher probability of finding an intermittent defect. Second, the script can loop through the pages while diagnostics are running. Third, comments can be added to the script to document dates, times, and other pertinent information to help trace the intermittent defect.

To date, we have used Selenium successfully during the build process of three major and three smaller systems.

Script Check List

We incorporate the following tests in our regression scripts:

1. Test all paths. A list or diagram should be developed as part of test planning then the script(s) are checked to verify all paths are tested.
2. Verify static text and field labels are present.
3. Test dynamic pages (fields are displayed based on previous entry or action). Verify elements are visible or not visible, as appropriate, for each dynamic variation.
4. Test for mandatory fields. Verify error messages.
5. Test for maximum lengths. Verify error messages.
6. Test for trimming in text fields (i.e., removing spaces at the beginning and end of an entry).
7. Test for allowable character set in text fields and for exact number of characters per field specifications. For example, nine digits in a Social Security Number; no area code starting with 0 or 1; *etc.*
8. Test for inter-field edits:
 - a. City, State, ZIP Code, Country validation
 - b. No Address Line 2 without Address Line 1
 - c. No Middle Name without a First Name
 - d. No Phone Extension without a Phone Number
9. Test for duplicate entries (e.g., two owners with the same Social Security Number).
10. Test Business Rule Validations (e.g., if the ZIP Code is within the TriMet area, transit tax information must be entered). Normally, these are specified in the use case.
11. Test alternate methods of entry (e.g., phone numbers with or without spaces or dashes).
12. After all field validation tests are done, complete all fields on all pages with realistic data or field-identifying data.
13. Test dropdown lists: all selections are present, default is displayed (if any), selection can be made.
14. Test dynamic dropdown lists (those where names, addresses, or phone numbers that were previously entered are added to the dropdown list for future selection). Test adding, updating, and deleting. Verify this functionality follows business rules (e.g., an Incorporator Name is added since it may be selected again, but a Bank Name is not added to the dropdown since it's unique).
15. Test navigation and back button protection on each page.
16. Test side bar, header, and footer links.
17. Test all interfaces:
 - a. To the Identify Management System (login and password).
 - b. To expert systems for determining business activity type and business name availability.
 - c. To the Credit Card Processing System.
 - d. To all workflows based on status and user role.

- e. To all back-end legacy systems via Web service calls.
18. Test transaction life cycle (i.e., test all states in the state transition diagram):
- a. User enters, stores, updates, deletes, and submits.
 - b. Agency rejects. Verify all error messages can be selected.
 - c. User can read all error messages selected. User can fix and resubmit.
 - d. Agency updates and accepts.
 - e. Back-end system is updated (full cycle).
19. Test validation at each step above and verify review and confirmation pages are correct.
20. Test user roles, verify authorized functions can be performed and unauthorized functions cannot be performed.

Maintenance

The following process helps us keep scripts up to date and minimizes the drudgery of maintenance. When a new build is about to be deployed, we review the release notes to determine the effect of the changes or new functionality on the scripts. We run the scripts to the point of change and then turn on Record Mode and start testing. Selenium records the tests, the script is updated, and the new version is saved. We run the complete script again to verify there have not been any other defects introduced and to verify the script runs with no errors or stops. We archive the previous version and document the changes made in the current version.

We use this process for deferred maintenance. If a defect is found and the team decides to defer the fix to a future release, we add a comment and a bypass in the script. When the defect has been fixed, the comment and bypass are removed and the original test is run. It does not have to be scripted again and the exact test that found the original defect verifies it has been fixed and does not reoccur. The test becomes permanent.

Evolution of the CBR Test Suite

When we started using Selenium in 2007, Central Business Registry (CBR) allowed our customers to file and pay for an Assumed Business Name (DBA) online. It has grown considerably, as the table below shows:

Measures of Oregon's Central Business Registry (CBR) System		
Measure	April 2007	April 2009
Number of Paper Forms Replaced	1	5
Number of State Agencies involved	1	3
Number of Web Pages	22	47
Number of Dynamic Pages	3	15
Web Services	3	8
XML Schemas	1	3
Transaction State Transitions	11	30

CBR's ultimate goal is to be a single site for a business to perform all government registrations, renewals, get licenses and pay fees. It now supports online filing of Assumed Business Names (ABN's), Domestic Business Corporations (DBC's), Limited Liability Companies (LLC's), and Domestic Nonprofit Corporations (DNC's) with the Oregon Secretary of State, along with a Combined Employer Registration (CER) that sets up an account at the Oregon Department of Revenue and the Oregon Employment Department.

These dates show the evolution of our test automation:

May 2007	First script, used primarily to support development.
August 2007	First full cycle regression script for ABN's.
November 2007	Added CER to ABN full cycle regression script.
February 2008	Full cycle regression script for LLC/CER.
May 2008	Full cycle regression script for DBC/CER. First with flow control.
October 2008	Duplicate Entry/Single Signature testing regression script.
January 2009	Full cycle regression script for twelve CER's.
February 2009	Full cycle regression script for DNC/CER.
March 2009	Four debugging scripts: ABN, LLC, DBC, DNP.
May 2009	Miscellaneous pages, paths, and states regression script.

From February 22, 2008, to May 20, 2009, there have been fourteen production builds. The average build cycle is 31 days. There have been 130 builds in our development environment; regression scripts were run for about a third of these. This is usually the environment where script maintenance and the development testing described above were performed.

There were 98 builds in our quality assurance (QA) environment and regression scripts were run for all of these. During this time 317 defects were discovered and fixed, and 153 enhancements were added to the system. There was one emergency build due to a defect reaching production. In the prior year there were three.

Limitations We Have Encountered

All testing tools have limitations. Our initial list was large due to our lack of experience. Here is our current list and what we have done as work-arounds.

Selenium IDE runs in Firefox, so running scripts in the IDE does not test the Microsoft Internet Explorer (IE) or Safari browsers. We perform our ad hoc user testing in IE and Safari to look for problems unique to these browsers. We plan to implement Selenium RC (Remote Control), which can run our scripts at the server, making it possible to test any browser.

Selenium IDE runs on the client. In order to run concurrent scripts, we use three PCs. Our start-to-end time for CBR, our most extensive test suite, is about 3 hours. After we implement Selenium RC we may consider the use of Selenium Grid, which allows multiple servers to execute scripts.

Extensions require Java Script. Java Script can be used directly in the script or to build actual stored commands. We use Java Script for:

- getting the date and time and appending to any field that must be unique;
- a stopwatch function;
- converting text to upper case;
- finding a link in a table where the link name varies;
- testing the browser back button; and
- testing trimming (removal of spaces).

We have not been able to test field-to-field navigation. When you execute

```
| type | firstName | John
```

Selenium looks for the element "firstName" and enters the text "John". Selenium does not know where the element is in relationship with other elements. We rely on our ad hoc testing for this. We will have the ability to address this when we implement Selenium RC.

There is no Find or Replace command in the IDE. We use comments to identify pages, sections, subroutines, etc., making it possible to find out where you are in the script. A text editor is used for large global changes.

Testing functionality outside the browser must be done manually. To test a printed report, enter an alert and pause. The alert provides instructions for the manual step that needs to be performed (e.g. click here and print report now). The pause gives the tester time to click Print in the Windows print menu and the script resumes at the expiration of the pause. We avoid this when possible to reduce manual monitoring

Observations

Time and Resources for Creating Scripts.

It is much faster to develop a test script in Selenium compared to our earlier hard-copy format. The time to develop a script in Word for an average use case was from 8 to 16 hours. This included reading the use case, running the applications, and writing out the instructions and expected results. This also included incorporating the use case into a script with other use cases and testing the resulting script by following each step and updating as necessary.

With Selenium, the time to script the same use case averages 2 hours, and we tend to be much more rigorous and add many more tests than we would in a Word script. Thus, we script four times faster with an increase in testing accuracy with Selenium.

This appears to be unrealistic until you look at our CBR application. In two years it has tripled in size and complexity, yet we are using the same resources for testing and have improved the quality of the releases.

Time and Resources to Run Scripts.

We can run a script in an hour that would have taken over 20 hours manually. This included an hour for a test manager to prepare and schedule the test, and an hour to review and document results. The remaining time was for testers to run the scripts. Running automated tests is 20 times faster than running the same tests manually and reduces manpower by a factor of 20.

With automated testing, the scripts can be started as soon as the new build is deployed. With manual testing, days could go by just getting the testers to start running scripts. After the script is run the log is examined for any errors. There is often a lengthy process of trying to re-create the error. Problem determination is much faster because the script can be stopped before the point of failure and stepped at that point. Automated testing reduces the testing cycle time from build to problem determination.

Testing Quality Increases

There is time for more planning; the tests are designed. The tendency is to be more thorough when creating an automated script. There is incentive to do it right since the test is going to be repeated hundreds of times. Besides, it's fun.

The script may have to be run several times if several builds are required before a release is ready for production. This is where manual regression starts to break down as mentioned above. With automated testing, a complete rerun of all scripts is not only feasible, it becomes routine.

Test Automation During Development

This has been the most valuable use of Selenium. It has allowed us to provide unit and functional testing concurrent with development. This interactive regression testing keeps the development from slipping backward. As the application matures, the script matures and the result is a regression script.

Script Maintenance

Script maintenance is costly and often is the reason manual scripts, as well as automated scripts, are abandoned. Selenium IDE records the maintenance testing so the script is automatically updated

Running scripts can also detect undocumented changes. If there is an undocumented change (not contained in the release notes), the script will often detect it and generate an error. Was this an omission in the release notes or an unauthorized change? Regardless, there was a documentation failure that must be addressed.

Testing Automation and Agile

Since the scripting process and testing is so much faster, we are far more agile. We are not at the stage of test-driven development, but our automated scripts flush out both development and requirement issues much earlier in the

process. Based on our CBR application, we are on a monthly Scrum cycle and have delivered significant new functions (153 enhancements over 14 months) with almost error-free code.

Return on Investment

Our return on investment is significant. Since Selenium is open source and the time to install is minimal, the only investment costs are (1) the time to learn the tool, (2) time to build scripts, and (3) extra PCs to run them. In our situation, our regressions scripts evolved from our agile development so there was not a separate scripting cost. That leaves training as the only significant investment. It took one tester a month to become productive and six to become proficient. The savings in manual script preparation and running has been offset over four times.

User Scripting

We have been successful in training several non-IT staff in scripting with Selenium. Our cashiering department manager scripted and tested a stand-alone credit card processing application, including all transactions, administrative pages, and role testing, in 2 days. Writing and running manual scripts would have taken 2 weeks.

Personnel Issues

Many of the personnel management issues are eliminated with automated testing. Selenium has no feelings that we have been able to detect. It can work long hours and does not interject opinions (we will talk more about opinions in ad hoc testing). It is as fast, accurate, and meticulous as the scripter permits. It does not whine when you need to run a script repeatedly. You don't have to bring it candy or buy it lunch.

Ad Hoc (Exploratory) Testing

Test automation has improved our ad hoc testing and vice versa. We do not need as many user testers, so we can use the most meticulous, pernicious, irreverent testers that have a knack of sniffing out bugs and breaking things. It has freed our testers from the script so they can truly run their own test scenarios and try the 'dumb things' they have observed. They can also test things "that will never happen" because they have the time and are not bound to testing the mainline functionality that is now in our automated scripts.

Our automated testing benefits from ad hoc testing through the use of test failure reviews. When ad hoc testers find a bug we conduct a review to determine how we missed it in our scripts. An ad hoc tester found a new validation that resulted in a trap. We discovered the validation was in the release notes, but the scripter missed it. Ad hoc testing caught it and our failure review pointed to the need for a double-check of the release notes.

Ad hoc testing benefits from the use of debugging scripts that can position testers at particular places in the application quickly. In CBR it can take 15 minutes to submit a registration before it appears in a reviewer's work queue. Once the registration is approved or rejected, it disappears from the queue. If the user wants to test that page, the script can generate any number of test registrations in a few minutes. The experienced tester can then focus on testing, rather than on data entry.

Ad hoc testing can also identify how we can improve the application. The drudgery and tedium of running manual scripts tends to stifle creativity. With manual scripts we would get little feedback from the testers other than the check marks in the Actual Results column of the script. Now our testers have time for suggestions to improve usability and reduce confusion.

Ad hoc testing is necessary for feedback crucial to improving quality and testing automation frees testers to provide that feedback.

Dual Monitors

This may sound like a minor point, but scripting and running tests with a single monitor is very difficult. You really need one monitor for the browser and one for the script.

No False Positives

In over two years of using Selenium, we have not seen any bug that was due to the tool.

Documentation, Help, and Training

The documentation for Selenium has improved considerably in the last two years. OpenQA is developing an excellent user manual. The OpenQA forum site is listed, as well. The Selenium site lists several firms that are addressing training and consulting. Formal training or a mentor would have helped our effort considerably.

Other Uses

The tool has many uses besides regression testing. Several have been discussed above and are recapped here along with a few others:

- Everything up? We have a script that can be run after an infrastructure change (e.g., a Solaris patch). The script opens all Web applications and performs a query in each.
- Demonstrations and Training. Anyone can demonstrate an application by just clicking pause/resume or running in step mode.
- We use “happy path” scripts with switches and stops for debugging.
- Help Desk staff are trained to use the “happy path” scripts so they can stop the application where the caller is having a problem. Then they can walk through the page, field-by-field and click-by-click. Prior to this we used hard copy screen shots for each page which did not show navigation or validation.
- Where’s the Slow button? We have used Selenium to execute one or more transactions with a stopwatch function to tune database settings and look for code or interfaces that may be slowing response time. We have scripted different combinations of search criteria that can be run easily and the response time measured for each. Then we can modify the query to see the effect.

Future Directions

Some of the enhancements that we are planning for the next two years include:

1. Implement Selenium RC and possibly Grid. We will move our most stable, hardened scripts to this environment to reduce the time to run a suite of regression scripts. Since we can integrate Java with the script, we can test some things we cannot currently test. Some examples are navigation within a page, ability to drive the script with a list or file, and save and analyze log files.
2. Continuous integration of the build process, running regression scripts and build deployment, if successful.
3. Measure actual code coverage by our Selenium test scripts. We have no internal measurement of code coverage by Selenium. We would like to implement a tool that would show untested areas of code.
4. Implement automated testing for all Web-based applications to the same degree. We have more extensive scripts and testing in some applications, and would like to extend the benefits to all.

Conclusions

Test automation can be implemented in small shops and provide huge benefits. We are a twenty-plus person shop with a testing department of one. Selenium has provided a path to automation and we have reaped significant benefits.

Test automation can eliminate the Regression Test Dilemma. Good practice calls for a full regression after any change, but there is never enough time to run one. Quality suffers because there are parts of the system that were not tested and may have bugs. We currently run full regression scripts after every build. The process has uncovered bugs in areas we would not have suspected and typically not directly associated with the changes.

Automated testing can be used throughout the development cycle with significant benefit.

Automated testing does not replace manual testing. Automated testing makes manual testing more effective.

References

Home site for Selenium that includes documentation, download, support, forums, *etc.*: <http://seleniumhq.org/>

Home site for the Oregon Secretary of State: <http://www.sos.state.or.us/>

Web Security Testing with Ruby and Watir

James O. Knowlton
james_knowlton@mcafee.com

Abstract

To verify the quality of web applications today, security testing is a necessity. But how to cover it all? SQL injection, cross-site scripting, buffer overflow...and the list goes on. Automating some of this testing would be great, but where to start?

This paper is a case study in how McAfee decided to use Ruby/Watir to help with it's Web security testing needs

The Ruby language, combined with the Watir module, is a great toolset for security testing of web applications. There are three reasons for this:

- Many of the common security vulnerabilities related to web applications (SQL Injection, cross-site scripting, and buffer overflow) have to do with simply posting different types of information to a web server via a client. This is pretty much what Watir is all about. It even gives you access to hidden elements, so its really is a great tool for submitting form data to a web server.
- The Ruby side of Watir, being a full-service language, has great tools for querying the database, checking audit logs and the like. Also, you can generate random data (or large datasets) to throw at a web app, or even pull the test data from a CSV file.
- There are some things you might not be able to do through Watir, but can certainly be done with Ruby. Again, this is perfect – because Watir is not really a test framework, it's just a way to drive the Browser when you need to. So, tests which are more low-level (such as web service communication or network tests) can be run through Ruby and RSpec, or whatever actual test framework you're using.

Biography

Jim Knowlton is a QA Automation Engineer with McAfee Inc., where he drives test automation efforts on McAfee ePolicy Orchestrator. He has over 18 years' experience in the software industry, including clients such as Novell, Symantec and Nike. He has been a Technical Support Engineer, a Technical Writer, a Beta Program Manager, and some other jobs he can't remember. He has a Bachelor's Degree in Management and is currently pursuing a Master of Business Administration degree at the University of Portland.

Jim is the author of "Python: Create, Modify, Reuse" (Wrox, 2008), and has also written website technical content for America Online, Amazon and IBM. He blogs at www.agilerubytester.com.

Introduction

We're going to talk about using Ruby, along with Watir, in building a security testing framework for web applications.

This paper will consist of the following:

1. A description of the problem we were trying to solve at McAfee
2. The possible choices we had and why we chose to implement our tests with Ruby/Watir
3. A quick overview of Ruby and Watir, and why we felt it would be especially good for what we were trying to accomplish
4. Demo of a few running examples of cross-site scripting and SQL injection tests running against my blog.
5. A review of our experience implementing the Ruby/Watir test framework
6. Lessons learned from our experience

Some security terms

To understand our challenge and how we approached building a solution, it is important to understand some basic types of security attacks applied to a web application:

- SQL injection - A SQL injection attack is one where an attacker takes advantage of the fact that information is passed from the web server to the database, and that the web server has access to the database. The attacker passes SQL commands in the input fields of a web application, and if the attack is successful, those SQL commands are then executed on the database server.
- Cross-site scripting - A cross-site scripting attack is one where a user embeds JavaScript in information submitted to the web server. That JavaScript would then be stored, and if another user brought the page up with that saved information, the JavaScript would then be executed. This can allow an attacker to get another user to go to a different website, reveal their username or password, or any number of other activities.
- Buffer overflow - A buffer overflow is simply an attempt to overflow the ability of the web server (or database) to process information. It consists of sending a huge amount of data to the web server. Often the goal of a buffer overflow attack is not to compromise a system, but to create a denial of service (make the system no longer available to others).

The challenge – testing security of a web application

McAfee has a web-based security management platform called ePolicy Orchestrator. It allows users to log in and perform various operations on client machines in a network, and run reports on the state of those machines.

Our challenge was to execute security tests (especially SQL injection and cross-site scripting tests) in an automated fashion, on all builds delivered to QA. As we saw it, we had the following requirements:

- The tests needed to be *automated* – we needed to be able to “push a button” and run all the tests, or subsets of the tests, and get results, with as little user interaction as possible
- The tests needed to be *easily developed and understood* – we wanted to develop the tests in a language and infrastructure that would be easy to understand and learn, since we would possibly be handing our tests off to other groups to run.
- The tests needed to be *license-free* – since we would be handing the tests off to other groups, and we had no idea if they would have licenses for any licensed technologies, we wanted to use open-source technologies to insure anyone would be able to set up a test environment.
- The tests needed to be *in a widely supported environment* – since development resources for support would be very limited, we needed to choose technologies for our test environment that are widely used and that have active communities (newsgroups, web sites, etc), so that we would have places to go should we run into issues (which we certainly would).

Choices and possible solutions

The following are two possible solutions we looked at, but decided not to pick.

Java/JUnit/Ant/WebTest

Since our application is written in Java, this is the first solution that occurred to us.

Advantages:

- Developers could give advice, since they all know Java
- Could directly access product classes
- Stable, mature architecture
- Development tool support - IntelliJ (IDE) and TeamCity (build automation tool) both support Java fully
- Canoo WebTest - We knew we would, for some tests, need something to drive the browser. WebTest is a good tool with an active community.

Disadvantages:

- Not much Java knowledge inside QA
- Since it is the same language as the application, there might sometimes be temptations to "hook into" the code and mock things when we really should be testing at the application level
- Building Java-based test frameworks tend to be more complex (sometimes necessarily so) than a scripting-language-based framework.

Python/unittest/batch files/PAMIE

Another option was to build a framework with Python, a popular scripting language. We broke down this option thus:

Advantages:

- Scripting language - Python is certainly easier to learn to be productive in than Java, for QA engineers who are not programmer-types
- Mature - Python has been around a long time, and there are tons of libraries and modules for it
- I'm very familiar with it - I've written a book on Python, so obviously I like it and am pretty familiar with it.

Disadvantages:

- PAMIE - PAMIE, which is a Python-based tool to drive internet explorer, is not often updated and lags behind open-source competitors in features.
- No build configuration tool - there really isn't a Python-based build configuration tool that is as mature and "used" as Ant (Java) or Rake (Ruby).
- Little integration with existing tools - IntelliJ has a Python module, but it's fairly basic, and there really is no Python support in TeamCity.

Ruby/Watir – why we picked it

We decided to go with a framework based on Ruby, RSpec, Rake and Watir. Why? Let's look at each technology in turn.

Ruby

Ruby is an object-oriented, open-source language that is very popular in the testing community. It is pretty easy to learn and there are a multitude of books, websites and online communities devoted to it.

Another advantage of Ruby is that our IDE, JetBrains IntelliJ, has a great Ruby module with full support for Ruby, RSpec and Rake.

RSpec

RSpec is a Ruby-based test framework. I chose RSpec for two main reasons:

- Test names can be written out in english, making test reports much easier to read
- RSpec can produce a really nice HTML test report:

```
RSpec Code Examples
3 examples, 1 failure
Finished in 0.172 seconds

sample expectations

  should pass with flying colors

  should also pass

  should fail

    expected #<Fixnum:9> => 4
    got #<Fixnum:7> => 3

    Compared using equal?, which compares object identity,
    but expected and actual are not the same object. Use
    'actual.should == expected' if you don't care about
    object identity in this example.

    .\sample.rb:11:

    9   end
    10  it "should fail" do
    11    (1+2).should equal(4)
    12  end
    13  end
```

Rake

Rake is the build configuration tool for Ruby. It has the following features:

- Rakefiles are actually Ruby scripts, so you have the full power of Ruby within your rakefile
- Rake has targets for RSpec tests, so it was easy to write test targets in Rake for RSpec
- Rake is fully supported in JetBrains TeamCity, our build automation tool.

Watir

We knew that for some of our tests we would need a way to drive the browser (there weren't API hooks in place to access product functionality). The advantage of using a COM-based tool which actually drives the real browser, rather than simulating a browser (as a tool like Canoo Webtest does) is that anything the browser can do, you can do automatically. Thus, it fully supports Ajax and Javascript, whereas WebTest "mostly" provides support.

Watir is the most mature and stable of the COM-based IE drivers, and it has a significant community behind it.

Some example security testing scripts

SQL injection

The following script is an example of a script to test for SQL injection. SQL injection is an attack technique that consists of embedding SQL commands in the entry fields of a web page. So, the basic way to test for this is to embed SQL commands in the web page to submit, then connect to the database and verify that the SQL command did not get executed. See the comments below (lines preceded by a '#' to follow the flow of the script:

```
#require needed modules
require 'spec'
require 'watir'
require 'dbi'

describe 'SQL Injection' do
  #initialize required variables
  before(:all) do
    username="testuser"
    password="testpassword"
    url="http://www.mywebapplication.com/login"
  end
  #first testcase, for sql injection, command to drop a table
  it 'should not be allowed when entering login name - drop table' do
    #start IE and go to login page
    ie = Watir::IE.new()
    ie.goto(url)
    #enter SQL command for login name and submit
    ie.text_field(:id, "username").set("'drop table reallyImportant'")
    ie.text_field(:id, "password").set(password)
    ie.button("Log On").click()

    #connect to db and check that table was not dropped
    connectstring="DBI::ODBC::mydbserver"
    db=dbi.connect(connectstring, "sa", "dbpassword")
    tablelist = db.tables #returns a list of db tables
    tablelist.should include("reallyImportant")
    db.disconnect()
  end
  #second testcase, for sql injection, command to create a table
  it 'should not be allowed when entering login name - create table' do
    #start IE and go to login page
    ie = Watir::IE.new()
    ie.goto(url)
    #enter SQL command for login name and submit
    ie.text_field(:id, "username").set("'create table junk\ (varchar 20\)\ )'")
    ie.text_field(:id, "password").set(password)
    ie.button("Log On").click()

    #connect to db and check that table was not dropped
    connectstring="DBI::ODBC::mydbserver"
    db=dbi.connect(connectstring, "sa", "dbpassword")
    tablelist = db.tables #returns a list of db tables
    tablelist.should_not include("junk")
    db.disconnect()
  end
end
end
```

Cross-site scripting

The following script is an example of a script to test for cross-site scripting. Cross-site scripting is an attack technique that consists of embedding JavaScript commands in the entry fields of a web page. So, the basic way to test for this is to embed JavaScript in the web page to submit (in this case JavaScript to generate a popup), then verify the JavaScript was not executed. Note that in the example below, I am assuming there is a `check_for_popups()` method - to actually check for Windows popups, take a look at the Win32OLE/AutoIT module in Ruby documentation. See the comments below (lines preceded by a '#') to follow the flow of the script:

```
#require needed modules
require 'spec'
require 'watir'
require 'dbi'

describe Cross Site Scripting' do
  # initialize required variables
  before(:all) do
    username="testuser"
    password="testpassword"
    url="http://www.mywebapplication.com/login"
  end
  #first testcase, for XSS string 1
  it 'should not be allowed when entering login name - XSS string 1' do
    #start IE and go to login page
    ie = Watir::IE.new()
    ie.goto(url)
    #enter XSS string for login name and submit
    ie.text_field(:id, "username").set("<script>alert('XSS');</script>")
    ie.text_field(:id, "password").set(password)
    ie.button("Log On").click()

    #use WIN32OLE/AutoIt to check for Windows popups
    check_for_popups("Windows Internet Explorer").should == false
  end
  #second testcase, for XSS string 2
  it 'should not be allowed when entering login name - XSS string 2' do
    #start IE and go to login page
    ie = Watir::IE.new()
    ie.goto(url)
    #enter XSS string for login name and submit
    ie.text_field(:id, "username").set("<BODY ONLOAD=alert('XSS')>")
    ie.text_field(:id, "password").set(password)
    ie.button("Log On").click()

    #use WIN32OLE/AutoIt to check for Windows popups
    check_for_popups("Windows Internet Explorer").should == false
  end
end
end
```

These scripts could be improved...

- Move common functionality into helper modules
- Extract data out and put in yaml (properties) file

Other types of tests

You could also use this framework for other types of web security tests (as we have), such as:

- Buffer overflow - Enter large amounts of data in form fields, and validate you get an appropriate error and no inappropriate errors on the server.

- URL manipulation - Have a list of product URLs. Feed the URLs one by one to a method that simply goes directly to the URL. Validate you get a login prompt.
- Information disclosure - Use Ruby directly to open log files and other system files; make sure there are no passwords or other sensitive information stored in plain text.

Our experience

We are very happy with the choice we made of using Ruby in conjunction with other tools to build an automation framework for security tests.

We used these tests in conjunction with HP QAInspect, which does a much more exhaustive, "carpet bombing" test of a web application. We found that the two approaches were more complimentary than duplicative:

- The Ruby tests had the advantage of actually checking the database for SQL injections, whereas QAInspect merely inspects to see where there might be vulnerabilities to SQL injection.
- The Ruby tests tell us precisely what is being tested and what the response is, whereas QAInspect (at least out of the box) is not as precise.
- QAInspect can span the entire application in a way that scripting tests by hand could never approach.

Some metrics

- Total security tests written - about 300
- Time to write the tests - about 1 month (we were working on other projects too, and it took a little longer than a month, so I'm estimating the "full time" time cost)
- Time it takes to run a full security test suite (all 300 tests) - a little over an hour

Lessons learned

We feel like we learned the following lessons from this experience, which was mostly a positive one:

- Have a constant focus on what technology and process will produce the best results, the fastest. Always focus on your central mission in QA, which is to provide accurate and timely information on product quality. Sometimes we can fall in love with tools or technology and forget why we're using them.
- Be on the constant lookout for ways to improve your process, as well as technologies to make things better.
- Part of "agile testing" is taking the initiative to communicate constantly what you're doing and ask management "does this provide you with valuable information? If not, what would?" We got lots of good ideas for test development from developers.
- Don't ever be emotionally married to your creation. If tomorrow we found something that worked better for testing our web applications, then we'd drop what we have. Remember - it's a program, not a child.

Final thoughts

We found Ruby, RSpec, Rake and Watir to be a great ecosystem for building web security tests (or any tests for a web application, for that matter). The combination of the power of the Ruby scripting language, the simplicity and cool HTML reports of RSpec, the flexibility of Rake and the ability of Watir to access the application directly through the browser makes this test solution flexible and powerful. Add in support for tools such as IntelliJ and TeamCity (Ruby is also supported very well in IDEs such as NetBeans and Eclipse), and you have a great tool for creating an effective automation framework.

References

Web Security

- Open Web Application Security Project (OWASP) - <http://www.owasp.org>
- SANS Institute - <http://www.sans.org>
- Foundstone WebSec101 online course - <http://www.foundstone.com/us/websec101.asp>

Ruby

- Ruby website - <http://www.ruby-lang.org>
- 'Programming Ruby' book online (1st edition) - <http://www.rubycentral.com/book>
- Ruby Essentials website - http://www.techotopia.com/index.php/Ruby_Essentials
- comp.lang.ruby newsgroup - <http://groups.google.com/group/comp.lang.ruby>

RSpec

- RSpec website - <http://rspec.info>
- RSpec API documentation - <http://rspec.rubyforge.org/rspec/1.2.8>
- RSpec Google group (mirror of mailing list) - <http://groups.google.com/group/rspec>

Watir

- Watir website - <http://www.watir.com>
- Watir API documentation - <http://wtr.rubyforge.org/rdoc>
- Watir-General Google group - <http://groups.google.com/group/watir-general>

Learning from Pragmatic Project Managers: Make Your Project Successful

Johanna Rothman
jr@jrothman.com

Abstract

You've managed projects but they're never quite right. They don't fit into the nice definitions in project management books. Your schedules are generally off. There's always some surprise. You're not failing, but you feel you should be more successful. Is there a solution?

Perhaps you can take a more pragmatic approach to project management. To manage the risk of not knowing what to do, use iteration to explore alternatives. To manage the risk of not knowing how far along you are, use incremental steps to finish features. To manage the risk of overrunning the schedule, make sure your project team and stakeholders agree on what "done" really means. Developers may think "done" means the code compiles cleanly. That's not how system users define "done." Another way to ensure success is to escape the dreaded trap of multi-tasking. It's a popular management style right now, but it drains time from everyone every time there is a task switch. Protect your staff and yourself with timeboxes—chunks of time when only one task is worked on. One final secret every project manager must know—there is no "one right way" to manage a project. Everything depends on your context, which includes the company and its products, the people on your team, and on you. Keep everything in balance and you'll have a successful project. But let something get out of balance, and kiss all your good work goodbye.

Biography

Johanna Rothman helps leaders solve problems and seize opportunities. She consults, speaks, and writes on managing high-technology product development. She enables managers, teams, and organizations to become more effective by applying her pragmatic approaches to the issues of project management, risk management, and people management.

Johanna publishes *The Pragmatic Manager*, a monthly email newsletter and podcast, and writes two blogs: Managing Product Development and Hiring Technical People. She is the author of several books:

- *Manage Your Project Portfolio: Increase Your Capacity and Finish More Projects*
- *Manage It! Your Guide to Modern, Pragmatic Project Management*
- *Behind Closed Doors: Secrets of Great Management* (with Esther Derby)
- *Hiring the Best Knowledge Workers, Techies & Nerds: The Secrets and Science of Hiring Technical People*
- *Corrective Action for the Software Industry* (with Denise Robitaille).

Learning from Pragmatic Project Managers: Make Your Project Successful

Johanna Rothman

New: Manage Your Project Portfolio, Increase Your Capacity and Finish More Projects

www.jrothman.com

jr@jrothman.com

781-641-4046



An Historical Perspective on Project Management



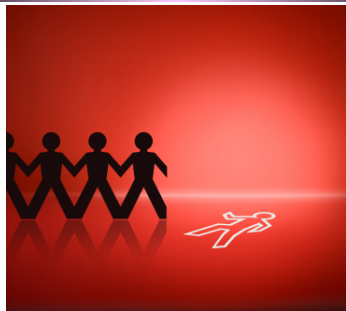
© 2009 Johanna Rothman

www.jrothman.com

jr@jrothman.com

2

Schedules Are Now Called Deadlines



© 2009 Johanna Rothman

www.jrothman.com

jr@jrothman.com

3

Software Projects Are Different...



© 2009 Johanna Rothman

www.jrothman.com

jr@jrothman.com

4

"Process Will Fix Everything"

The One True Way

© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 5

Not!



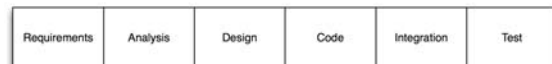
© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 6

What Does a Pragmatic PM Do?

- Choose a lifecycle that works
- Plan to replan
- Prototype where necessary
- Implement by feature before selecting an architecture
- Never multitask

© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 7

Serial Lifecycle



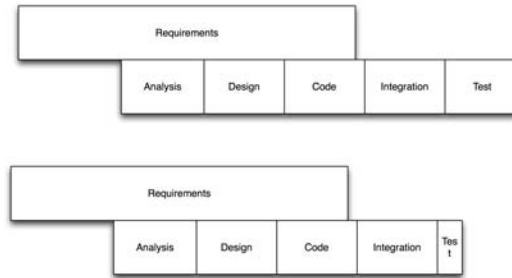
© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 8

Serial Lifecycles I Have Met



© 2009 Johanna Rothman www.jrothman.com jr@rothman.com 9

More Serial Lifecycles I Have Met



© 2009 Johanna Rothman www.jrothman.com jr@rothman.com 10

But, Change Happens



© 2009 Johanna Rothman www.jrothman.com jr@rothman.com 11

When Was the Last Time Just One Thing Changed?



© 2009 Johanna Rothman www.jrothman.com jr@rothman.com 12

Sometimes You Get Lucky



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 13

What More Frequently Happens



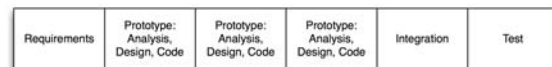
© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 14

As a Result



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 15

Iterative Lifecycle



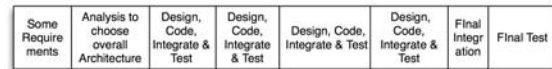
© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 16

Prototype and Learn From Experience



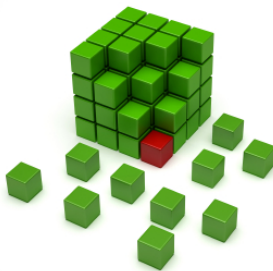
© 2009 Johanna Rothman www.jrothman.com jr@rothman.com 17

Incremental Lifecycle



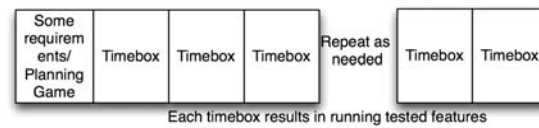
© 2009 Johanna Rothman www.jrothman.com jr@rothman.com 18

Pieces



© 2009 Johanna Rothman www.jrothman.com jr@rothman.com 19

Agile Lifecycle



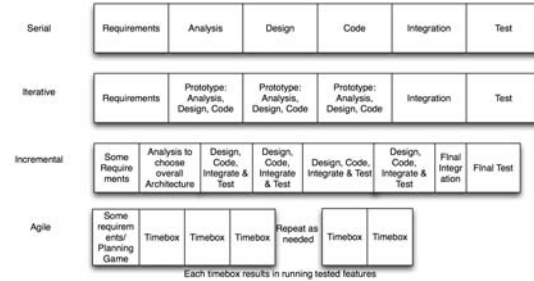
© 2009 Johanna Rothman www.jrothman.com jr@rothman.com 20

Any Project Can Use Chunks of Work in Timeboxes



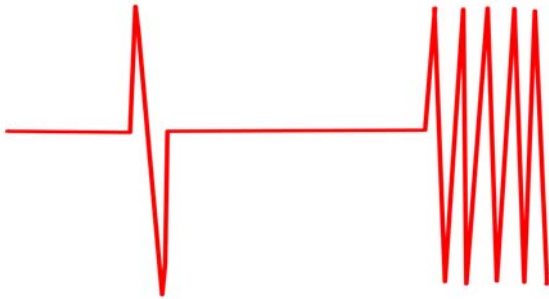
© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 21

Quick Look at All Lifecycles



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 22

Up-Front Planning Leads to Write-Once/Read-Never Schedules



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 23

Rolling Wave Planning



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 24

Prototype



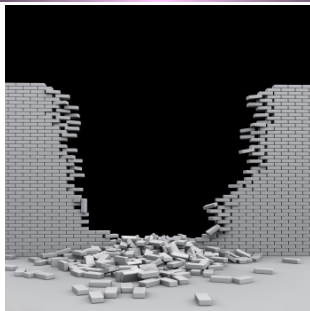
© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 25

Start with the Architecture Leads to Pretty Pictures



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 26

Implementing by Architecture Doesn't Always Make Features Possible



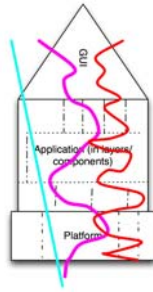
© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 27

Implement by Feature Helps You Check Features Off as Done



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 28

An Architectural View of Implement by Feature



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 29

You Start With One Thing to Do



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 30

You Take on More Work



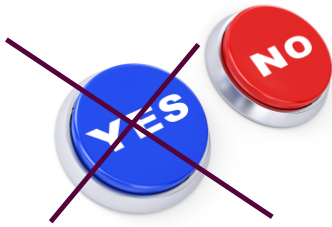
© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 31

Until You Crack



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 32

Say No



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 33

Some of My Lessons

- Use a lifecycle that matches your project's risks
 - Plan to replan
 - Learn from the work by prototyping and implementing by feature
 - Say No when asked to multitask
-
- Avoid those *deadlines*

© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 34

Questions?

- I write about these ideas on my blog Managing Product Development, jrothman.com/blog/mpd and in my newsletter, the Pragmatic Manager. You can subscribe on my site or fill out a yellow form here
- Take a look at *Manage It! Your Guide to Modern, Pragmatic Project Management* and *Manage Your Project Portfolio: Increase Your Capacity and Finish More Projects* for more ideas



© 2009 Johanna Rothman www.jrothman.com jr@jrothman.com 35

Can't Travel? Virtual retrospectives can be effective!

Debra Lavell
Intel Corporation
Debra.S.Lavell@Intel.com

Abstract

Tick, tock. Tick, tock. The clock is ticking. It is the end of a long SW project and the geographically dispersed team wants to spend time to look back and capture what worked well and what needs to be done differently next time so they can improve the next project. You need a process to gather learnings, but you have ZERO travel budget. What to do? This paper will explore specific key take-a-ways and Best Known Methods (BKM) you can use to prepare for and successfully execute a virtual retrospective.

Biography

Currently, Debra is the Organizational Learning & Retrospective Program Manager in the Corporate Platform Office at Intel Corporation. With over 10 years experience, Debra has delivered over 300 Project and Milestone Retrospectives for Intel world-wide. Prior to her work in quality, Debra spent 8 years managing an IT department responsible for a 500+-node network for ADC Telecommunications. Debra is a member of the Rose City Software Process Improvement Network (SPIN) Steering Committee. She currently is the President of the Pacific Northwest Software Quality Conference, Portland, Oregon. She holds a Bachelor's of Arts degree in Management with an emphasis on Industrial Relations. To contact please email: Debra.S.Lavell@intel.com

Copyright Debra S. Lavell October, 2009

Published at the Pacific Northwest Software Quality Conference 2009

Introduction

Tick, tock. Tick, tock. The clock is ticking. Imagine...a Vice President of a significant project has asked you to facilitate a “lessons learned” session. This team has been working together for over 18 months. There are three software development teams spread across several time zones (California, Israel and China). You have decided to try using the retrospectives method for capturing key learnings. You know other post-project audit methods are used within your organization. Many teams feel that they are completely adequate, besides most teams don’t have a travel budget to bring the whole team together for a three-day, off-site retrospective as Norm Kerth describes in his book *Project Retrospectives: A Handbook for Team Reviews*. Because of this, you know that you will need to modify the approach as you are anticipating resistance to this new methodology. How do you proceed?

Although there are many ways to introduce change into an organization, having a defined approach and plan is fundamental in order to overcome resistance to change and to gain senior stakeholder buy-in. This paper describes the approach that was used for introducing the retrospectives methodology into Intel, a large company with many distributed teams, as well as the factors that were beneficial in gaining organizational buy-in for broad deployment – without a travel budget!

Start with a Problem, Not a Solution

Many times change agents become so enamored with the latest process, tool or practice which they are championing, that they try to sell the solution to the organization for the sake of the solution. In effect, it becomes a solution in search of a problem.

Whether it is a new configuration management tool, risk management process, or project review methodology, we have learned that to be more successful, the approach has to be reversed. First a problem has to be identified, and *then* a solution can be applied. The advantage of this approach is that you are helping to solve someone’s business problem, while gaining support for your solution at the same time. Further, to increase the probability of solving the business problem, the change agent should tailor the solution to fit the organizational culture, processes, and application. If your company is anything like Intel, this is critically important in order to get over the “not invented here” hurdle.

In 2001, after reading Norm Kerth’s book I was so moved by the possibility of a better, more effective way to capture learnings that I decided introduce the method into Intel. It took a few months, but an opportunity to pilot the practice became available with a team who had invited our organization to help them improve their requirements engineering practices. The intent was to uncover what worked well on their last project, and to identify what the team wanted to do differently on the next project – which at the time was entering the requirements gathering and specification phase. Comments from the

team such as “our requirements sucked last time” prompted us to ask the project manager if she could get the team together to do a retrospective to find out what they really meant by “our requirements sucked” and what the business impacts were.

Intel has many globally dispersed teams; for example one software team was dispersed, with members in three time zones and getting them together face-to-face was becoming more and more difficult. After a kick-off meeting it could be 18 months before the team would be able to travel. In late 2008, Intel declared a “no travel” policy, as a way to save dollars and preserve jobs. Conducting retrospectives without a travel budget meant we had to re-think our approach to keeping the participants engaged and still gather learnings. Full day, face-to-face meetings were now a thing of the past, so we had to come up with a new strategy to uncover and gather the key learnings.

Begin Small and Establish Pull

When introducing a new idea into an organization, we have found that it is beneficial to begin with a small implementation, and expand once demonstrable results are achieved. We always try to begin with a proof of concept or pilot implementation so we can control the organizational variables involved. This “incubator” approach increases the probability of successfully achieving positive business results.

We followed this approach for retrospectives by approaching a single project manager within Intel who we knew would be open to trying something new and had a business problem to solve (requirements that sucked). We held several two-hour virtual meetings with small teams, over a telephone bridge. We used a collaboration tool (Live Meeting) to gather the team’s feedback on what happened on the previous project. The sessions were held during normal business hours convenient for the team members. The team in India spent an additional hour harvesting the data to identify primary sources of the problem and to develop action plans going forward. As the facilitator was in Oregon, which meant a late night for her, but the team was engaged because it was morning for them so they were fresh and ready to get their ideas captured. At the end of the meeting, she asked the team if they felt the time spent generating insights was valuable. The feedback from the team was overwhelmingly positive.

In fact, the project manager was so pleased that he agreed to sponsor additional retrospectives within their organization, and pitch retrospectives to his senior management team as a standard method of collecting and implementing project learnings. The senior management team was so intrigued by the results of the initial retrospective that they implemented mandatory end of project retrospectives for every project in their business group. Over the course of a year, we facilitated over 15 retrospectives within that business unit. From the data that was collected, common themes and improvement opportunities emerged which resulted in standardization of some key program and project management processes, such as good requirements engineering practices. Additionally, the retrospectives methodology was established as a standard practice by touching all programs, multiple times across the lifecycle.

What happened next was a bit surprising and exciting. Program and project managers from other business groups started to hear about the results that were being realized in the original business group, and wanted a piece of the action. “We want what they have” was a common statement from the other business groups. By demonstrating positive business results, we created pull for retrospectives at a much greater scale than had ever been anticipated.

Fast forward to 2009, we now have over 10 trained facilitators in India to support the requests. We conducted the training session using video conferencing technology where we could effectively train the new facilitators’ long distance, with the instructor in Oregon and facilitators in Bangalore. By starting small, establishing pull we now have an established process and trained facilitators which didn’t require any travel.

Tailor the Solution as Needed

As more and more teams were conducting post-project retrospectives (to improve their requirements processes) we came to realize that by facilitating a mid-cycle retrospective, for example, right after the requirements specification was complete it would allow a team to capture planning learnings as they occurred and when their memories were still fresh. This change in the retrospective methodology created the opportunity for a project mid-way through their planning phase, to benefit from the learnings and implement improvement actions. The program and project teams no longer had to wait until the next program or project to realize the benefits.

We have data to support we are able to achieve better results by conducting multiple retrospectives along the way, and implementing improvements into an organization faster. The standard retrospective methodology that we now use at Intel recommends multiple retrospectives at three strategic milestones (after requirement definition, after Alpha and after launch) during the course of a project lifecycle.

How did we do it?

We started by looking at the team’s recent milestone: Alpha. We wanted to make sure we catch the team and stop right after a major milestone to gather their learnings. We worked with the project manager to identify an already established team meeting where the key stakeholders regularly met to discuss the project. If it wasn’t already two hours, we asked to extend it so we had adequate time to review feedback gathered via a pre-work survey. We found going longer than two-hours was too difficult for those who were dialing in from another time zone. Since the meeting was already a “virtual” meeting it was easy to modify the agenda to focus on reflecting on what the team experienced to deliver their Alpha code. Below is an example of the email notice to attend the retrospective:

As part of continuous process improvement, we will dedicate next Wednesday’s normal Product Development Team meeting at 10am (Pacific Standard Time) to discuss what worked well and what we may want to do differently as a result of our Alpha release. We will use our learnings going

forward to improve the effectiveness of the team. Please note, I have extended our normal one-hour meeting to two-hours, so we can meet the objectives of a retrospective which is:

Increase the capability of the team by:

- *Sharing perspectives to understand what worked well in the project so we can reinforce it*
- *Identifying opportunities for improvement and lessons learned so we can improve subsequent projects*
- *Making specific recommendations for changes*
- *Discussing what the team wants to do differently*
- *Helping the team see ways to work together more effectively*

The next thing we did was negotiate with the sponsor the agenda. Below is an example agenda:

Agenda 7:30 – 9:30 pm (Pacific Standard Time)

When	What	Expected Outcome
10 min	Introductions/objectives & ground rules	Align on purpose of the meeting & enable an effective meeting
10 min	Quick Program Overview	Ensure we understand the scope of the retrospective, value proposition and timeline
60 min	Review survey data to identify key messages from the feedback	Break into smaller sub-teams to synthesize raw data into top learnings
30 min	Short report out from each break out team	Identify recommended focus areas & common themes
10 min	Wrap up and set next steps	Discuss how the team will create action plans with the owners

Notice 90 minutes of the two-hour meeting is where we break into smaller sub-teams and report out the key messages. How would we do that virtually? Since the meeting is on a telephone bridge, using a collaboration tool such as Live Meeting we would start on one main bridge line. Get started and then break into three teams (Team A, B and C) where we pre-assigned a new bridge number to dial into, a survey question or two for them to discuss offline with a pre-set facilitator and a timekeeper.

Here is an example of the email to recruit the facilitator and the timekeeper:

Hello,

I could use your help tomorrow during the Website Software Project retrospective.

Jason and Scott: would you be willing to lead a small break out team discussion during the retrospective? Your role is to help the team create a summary slide of the survey data.

Don't worry, this requires no preparation – we will be walking you through the process, however, we wanted you to be aware we will be asking you to:

- *Scott: drop off the main bridge and go to another bridge I have set up*
- *Jason: stay on the main bridge (see attached slides)
Lead the team to create a summary slide with key messages associated with the survey questions your team has been assigned (see attached slides)*
- *Email the slide(s) back to me so I can display it to the larger group*
- *Report out the roll up summary of the key messages from the discussions in your group*

Don and Jeff: would you be willing to communicate with me via IM as a timekeeper? Help me gauge how your team is progressing and if needed – let me know if there are any questions that need answers.

If you feel uncomfortable in these roles, please let us know ASAP so we can assign another person to help us tomorrow.

Thanks,

Once we had all the roles assigned and the process clear to everyone, the retrospective participants would drop off the main bridge, go to their assigned team and begin synthesizing the survey data. The sub-team facilitator would help them pull out the top 3-5 items they felt were important and the timekeeper would stay on IM with me to ensure they would email me their completed slide so I could display it once everyone came back to the original bridge. We would ask someone from each team to report out their findings and discussions with a broader audience to ensure we didn't miss anything important.

After the sharing of the key messages, the next step is to uncover emerging themes and prioritize what is important to the team. What is the team interested in spending time to go fix? Who has the energy and passion to spend an hour outside this initial meeting to write an action plan? Who else, besides the attendees should be included in the action planning, so we can ensure change happens?

We have found forming smaller sub-teams to work on action plans, after the two hour retrospective allows those with passion to spend time (one hour) to get to root cause and offer recommendations to fix the issue.

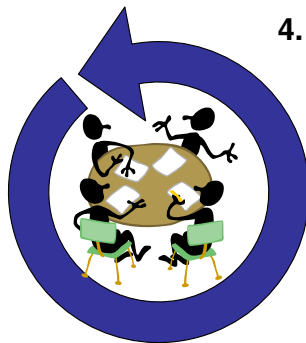
The last step, typically a week later, is to come back to the larger group to report out the action plans the sub-teams developed. Do we have any gaps? Did we identify the root cause? Are the recommendations complete? Will the broader team support the next steps? Can we break down the next steps into 2-3 week increments with owners and due dates so we can make forward progress?

Below is the four-step, closed loop process we have been describing:

The Four Phases of a Retrospective

1. Planning

- Plan the logistics
- Collect and organize project objective data
- Conduct Retrospective Survey
- Analyze survey data



4. Close the Loop

- Close Action Plans
- Document key learnings
- Affect changes in current (if applicable) & future programs
- Update lifecycle framework as needed

2. Retrospective Meeting

- Review Survey Data
- “Harvest” the data for common themes
- Identify volunteers to create action plans

3. Plan for the Future

- Sub-teams to write action plans.
- Assign owners & execute Action Plans
- Communicate outcomes to Management

By tailoring our approach we were able to work within the needs of the team. We learned:

- Use an already set meeting (don’t add an additional meeting to their already busy calendar)
- Focus on a specific milestone
- Break into smaller working teams to ensure everyone participates fully, both for synthesizing survey data and writing action plans
- Use a neutral facilitator so all perspectives are being heard
- We saved the travel budget

The feedback from the teams has been tremendously positive! Teams like the way we are able to cover a lot of ground in a short period of time by being creative with sub-teams and additional phone bridges. In addition, we use the technology to our advantage and no one had to leave their cube or comfy home office.

Gain Senior Management Support for Broad Deployment

Organizational buy-in is vital for the wide-spread success of any initiative, and senior management support is critical to achieving that buy-in. However, senior managers don't have to be intimately involved from the beginning. Allowing a new idea to gain support and momentum prior to exposure to senior management is a good strategy, as we have demonstrated. Normally, to gain wide-spread adoption (and organizational support) for the new idea, it **does** require the endorsement and advocacy of the senior leaders of an organization.

Since 2001, the momentum for retrospectives has bubbled up to the senior management level many times as a result of a program or project team sharing learnings and distributing action plans within a business group. During one project retrospective, where the project manager wasn't originally sure he wanted to do a full retrospective, the senior manager of the business group was invited to participate. As the end of the process, the senior manager stated that he liked the result of the "new" approach and wanted all teams to adopt this methodology. The funny part was that unbeknownst to the senior leader, the large majority of the teams in his organization were already doing some form of retrospectives.

Senior management support has allowed us to approach teams who had previously shunned the new practice. Each year we have facilitated more retrospectives. At one point, the pull for a skilled, retrospective facilitator was more than one person could keep up with. To accommodate the increase in demand, we established a full day training course to develop facilitators in each business unit. Our philosophy is that we want to train at least two program managers from each business unit so they can be a resource for the whole group. Growing the skill set within the business unit allowed more retrospectives to be conducted. To date we have over 120 trained facilitators at Intel, in several international locations with a growing need for facilitators in China, Russia, and Japan. Positioning facilitators in different parts of the world has decreased the cost for travel because we always try to use a local resource or at least on the same time zone to avoid any travel.

This scenario takes years to achieve. What if you don't have eight years to build momentum for retrospectives? How would you approach a senior manager who has not expressed an interest in retrospectives? We have found that the higher you go into an organization, the more important it is to be able to articulate the benefits of retrospectives to gain organizational buy-in. Showing senior managers that retrospectives can improve productivity, reduce costs, and increase efficiency all helped gain support for retrospectives across our company – and if it doesn't require travel, all the better!

Conclusion

The next time you are asked to collect lessons learned, try conducting a virtual retrospective! There are many advantages:

- Focus on current problems within the team or organization
- Start small, with a team who is open to trying a retrospective
- Show results on a pilot implementation, then others will come knocking on your door
- To accelerate the adoption, meet with top leaders to share results and educate them on the benefits of multiple retrospectives, spread across the lifecycle of a project

Rather than ask senior leaders to mandate “all programs and projects shall conduct a retrospective at three strategic milestones” we have found a more organic, bottoms-up pull for the practice has worked at Intel. This approach has proven to be our recipe for successful organizational buy-in which resulted in obtaining resources needed to persuade teams and business groups to adopt retrospectives as a common practice for capturing key learnings and driving improvement actions.

References

Kerth, Norman L., 2001, *Project Retrospectives: A Handbook for Team Reviews*

Retrospective Analysis and Prioritization Areas for Beta Release Planning Improvement

Ajay Jain

ajjain@adobe.com

Adobe Systems Incorporated

Noida, India

Abstract

The beta release of a product is an official pre-released version of the product. The product at this stage i.e. at the beta testing stage is generally considered to be fully functional and in a “close-to-release” state. Typically, the participants in beta testing are a small group of classified users who are invited by the product company. In some cases, however, the beta product is open for public scrutiny.

This paper does a retrospective analysis by taking a project and product as a case study and evaluates the testing effort invested by the testing team in beta or prerelease testing and makes a statistical comparison with that of the testing effort invested in regular feature testing during a complete product cycle. The data variables used are in terms of resources, the time spent in bug triaging, beta readiness certification, and documentation. The data is then analyzed to measure the effectiveness of the beta or prerelease testing program in terms of the product quality improvement achieved through the beta or prerelease testing program. Post-analysis recommendations are also suggested to help a beta testing program achieve a high quality product without overshooting the testing cost.

The strategies discussed in the paper shall help a test manager in achieving a much higher ROI (Return on Investment) from the beta release test program not only in terms of testing effort spent but also in improving the product functionalities and optimizing the organization cost.

About the Author: Ajay Jain

Ajay Jain has more than nine years of industry experience in testing and validation of mobile and desktop publishing software. He is currently working with Adobe Systems, Inc. as Quality Engineering Manager, leading and managing the Instantiation (Deployment, Provisioning and Services) QE team for Adobe Creative Suites products. Prior to Adobe, Ajay Jain worked with industry majors like Lucent Technologies (Bell Labs Development Center) and Skyworks Inc, where his experience ranges from building a startup testing team from scratch to a resource optimized, efficiency driven team certifying multiple product lines.

Ajay has an active interest in knowledge sharing on best processes and practices and has written and published several papers for *Quality Matters*, Adobe Quality Newsletter (Internal), and hosted birds of feather session at Adobe QE Summit. Two of his papers, “Power of Glide Path: Statistical approach for controlling and adding Predictability in a testing project” and “Well processed, well done – Suggesting 5 light weight processes for optimizing software test management” earned him speaker invites to the 8th All India Software Testing Conference 2008, and SPICE Conference 2009 respectively. He also has a patent invention–Meritorious Disclosure award to his credit.

Ajay holds a B.Tech (Engineering) degree from Delhi Institute of Technology and a Specialized Diploma in Business Administration from the Institute of Management Technology. Reach him at ajjain@adobe.com.

Introduction

Beta testing is a strategic milestone in a product life cycle and helps in building and securing a strong product positioning among the potential end users or consumers. Therefore, it is extremely important to plan and manage the testing effort in bringing a product to a beta testing stage.

Releasing a quality product on time, while restricting the cost to a minimum level, is a challenge for the test manager. She has to make sure that the right resources are invested at the right time on right activities. There is almost a zero scope for any testing effort getting wasted or going into unnecessary activities.

For a test manager, the beta testing phase involves complex planning and meticulous execution. A product's beta (a successful one) is a major milestone for any team and organization. When the product wins accolades and receives positive feedback from potential customers, it is also a big win for the testing team because a successful beta establishes the testing coverage and product stability achieved while doing the testing of the product.

Primary reasons for releasing a product for beta testing are:

- Platform to introduce the product and features to wide audience.
- Receive bugs and compatibility issues on existing functionalities from the end users.
- Receive feedback and suggestions on existing features and new features requests.

Testing is always a crucial, challenging, and interesting activity for a test manager especially if she is managing multiple time bound projects with a limited set of testers. She cannot afford to lose the quality of any project, she cannot afford to delay any project, and she cannot afford to burn out the team.

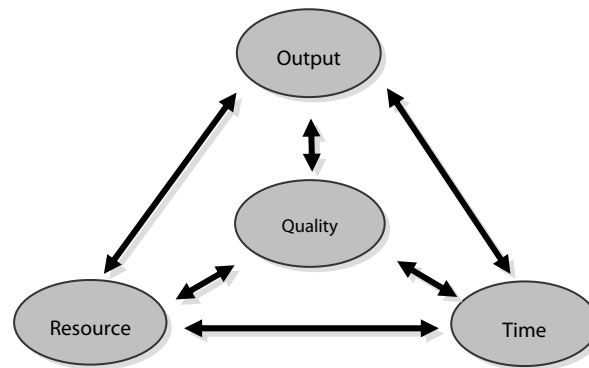


Figure: The Dimension of Productivity concept

This paper takes a case study of beta testing done for Adobe Creative Suites 4 family of suites. The manager managing the complete testing team owns validation of six products with seven head count of test engineers in her team. The testing cycle of each of these products is 12 months. She is now conducting a retrospective analysis of the last projects done and aiming for an effectiveness evaluation of the testing effort of the beta or prerelease testing program.

Time and effort investment in beta testing calculation

Beta testing involves the following major activities:

1. Beta Readiness Certification

Before selecting a build candidate for the beta release, the product or application always goes for a full testing cycle (with respect to offered features) within the internal team. This full cycle covers all stages starting from requirements validation to feature test case execution. A good strategy is to create a profile of the end user's machine, platforms, and the most likely products already installed and available on the user's machine. This strategy helps to design and observe the product behavior in application coexistence scenarios. An application coexistence scenario unearths several unexpected issues because of non-compatibility of two different applications. For example, the installation of Adobe Creative Suite (where Adobe Photoshop is one of the 10 products constituting the suite) on a machine where the Photoshop trial version is already installed might result in some kind of an installation conflict. A common area where conflict implementation is important and interesting is the coexistence of prerelease and trial versions with a full licensed version.

The readiness certification also includes validating major bug fixes that were reported on the feature. The aim of the certification is to make sure that the application is working fine on all major platforms and is sufficiently stable in all general and exceptional conditions. Once a substantial confidence is achieved on the product's functionalities, only then the product is selected as a beta candidate.

The test manager now does an effort calculation exercise to determine the effort that the test engineers have invested while certifying the beta build candidate.

Effort Calculation Category: Beta Build Readiness Certification (per Product, per platform)
Testing effort – 6 person days
Number of products - 6*

**Note: Test manager is managing six products in her team. The data above is based on the execution time for beta build test cases. The feature testing effort is considered part of the regular testing effort and is not included in the beta testing effort. The beta testing effort is the dedicated effort that the QE (tester) had put for determining the candidature of a specific product build as the beta build.*

2. *Documentation for beta testers*

Documentation for beta testers is generally referred to as release notes. A specific release note should be included with each and every beta build. Release notes should be prepared with utmost care because this is the first document that will help the user to get the product rightly installed on their machines. Any misinterpretation at the release notes level may result in the flow of unnecessary and invalid bugs from the tester's side.

A *release note* should carry the following information:

- a) Detailed installation or setup instructions
- b) Description of existing known issues, limitations, and other relevant information
- c) Hardware and software configuration, platform, system, and other technical requirements, and specifications for product installation or functionalities
- d) Available features and features recommended for testing
- e) Methods (Bug Database/Tools/Forums) for reporting bugs and issues
- f) Disclaimer*

**Because the product is being released only for beta testing, there is no guarantee of a bug-free product. There may be cases where the user's machine might face operation disruption, or BSOD (Blue Screen of Death) or other critical crashes originating might be because of the beta product. It is always advisable to have a Disclaimer section, clarifying no responsibility for data loss on the user's machine.*

Effort Calculation Category: Documentation (per product, per platform)
Testing effort – 0.5 person days
Number of products - 3

3. *Bug triaging*

Bug triaging consumes a major part of a tester's time. As soon as the product (or application) is out for beta testing, there is always a big influx of bugs from the beta testers.

Bug triaging includes the effort spent by the tester in:

1. Reviewing a new bug, categorizing (or re-categorizing) the bug against the right product areas in the bug database
2. Simulation of bug scenarios or failures in order to get the logs, especially if the bug description or log files are incorrect or incomplete
3. Update and closure of bugs with right details

Based on the average time spent on bug reviews, bugs are divided into 3 categories:

Category 1 bugs – These are bugs that take less than 10 minutes to triage; these bugs are generally not new from the product's perspective and are already there in the team's bug database. In our case of creative suites, the number of Category 1 bugs is 220.

Category 2 bugs – These are bugs that take close to 30 minutes for the complete review. Typically, these bugs have good bug description based on which they can be simulated or reproduced in-house, even without log files. In our case of creative suites, the number of Category 2 bugs is 130.

Category 3 bugs – These bugs take the maximum time (sometimes several hours) to simulate. The beta testers reported incomplete bug descriptions and did not attach log files. The in-house tester has to extract details from the beta testers on the exact reproducibility conditions. As a result, Category 3 bug troubleshooting is tough and time-consuming. In our case of creative suites, the number of Category 3 bugs is 50.

Effort Calculation Category: Bug Triaging (Including all prerelease/beta drops)
Total beta bugs reported: ~400 (Category 1: 220, Category 2: 130, and Category 3: 50)
Total bug triaging effort: $220 * 10 \text{ minutes} + 130 * 30 \text{ minutes} + 50 * 120 \text{ minutes}$
: ~12000 minutes.
Average Bug Triaging effort: (Total triaging effort / Total beta bugs reported)
: $\sim 12000 / 400$
: ~30 minutes per bug

The data is being rounded off to a whole number in some cases to simplify calculations.

4. Communicating with beta testers on forums/e-mails

Keeping the beta testers up to date on the status of issues, providing a quick clarification to their posted queries, is always desired from an effective beta program. Forums provide a platform where the beta tester community comes together. In the project discussed in this article, beta testers had added 250 posts on the beta forum. The average time spent by an in-house tester on a forum post was 5 minutes. However, some posts required around an hour of investigation and discussion. Also, the nature of participation was analyzed and it was found that only 5-7 out of the 50 beta testers were regularly posting questions.

Total Posts: 250
Average time spent (per forum post): 5 minutes
Total time spent on the forum response: $250 * 5 = 1250 \text{ minutes} = \sim 12 \text{ days}$

Assumptions:

- 1) *Time invested in management-level meetings is not added to the data estimation.*
- 2) *1 person day = 8 working hours*

Total effort invested in beta testing:

Table 1: Summarized view of testing effort invested in beta testing

Effort Calculation Category	Time Spent (in person days) (Per product / Per Platform)	Number of Products	Number of Platforms	Total Time (in person days)
Beta Readiness Certification (a)	6	3	2	36
Documentation (b)	0.5	3	2	3
Bug Triaging/Troubleshooting (c)				25
Forum Responses (d)				12
Total Time (a + b + c + d)				~75

Therefore, the total time invested on each beta or prerelease testing is 75 person days.

If a business unit or product team is planning for 3 beta releases, the time investment will be 75*3 person days i.e. 225 person days.

Now, let us do an analysis of the beta reported bugs/failures/issues. Here are the different charts derived from the ~400 bugs logged through the beta or prerelease program.

Chart 1: Beta program bugs classification as per their final closure state (Fixed/ Withdrawn)

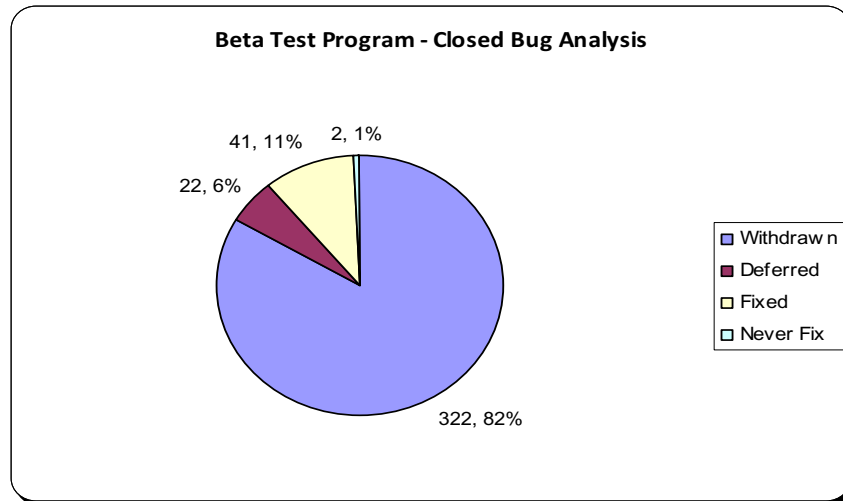
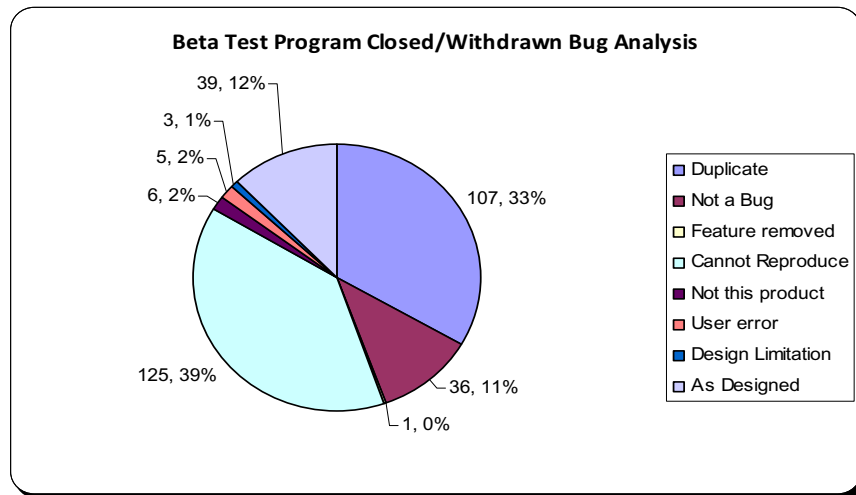


Chart 2: Beta program bugs classification as per their Closed-Withdrawn reasons



Here are the charts derived from the ~1400 bugs logged during the regular feature test cycle.

Chart 3: Regular feature test program bugs classification as per their final closure state (Fixed/Withdrawn)

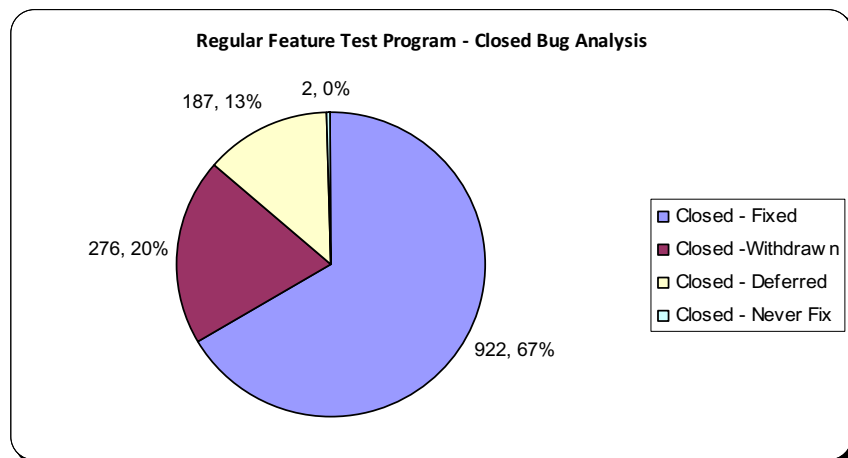
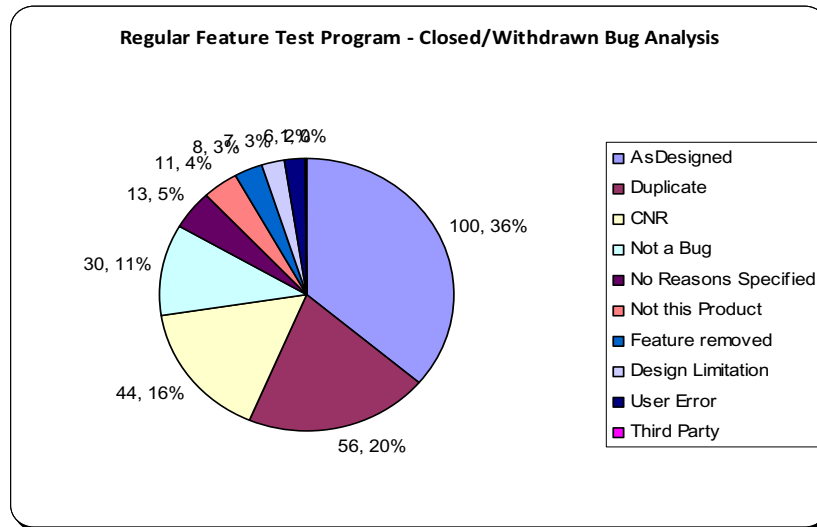
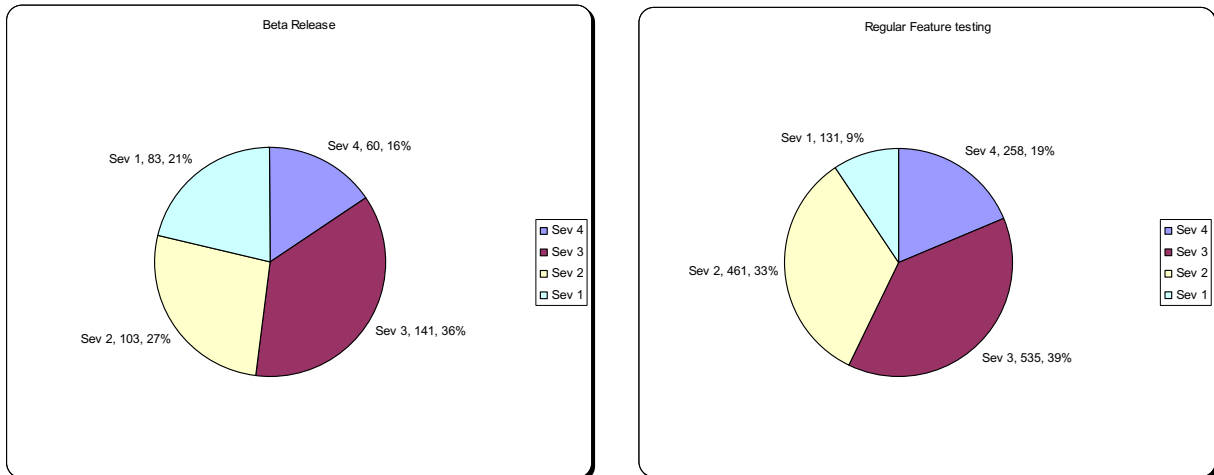


Chart 4: Regular feature test program bugs classification as per their Closed-Withdrawn reasons



CNR stands for “Can not reproduce” i.e. bugs which the core test engineer is not able to reproduce in house.

Chart 5: Severity distribution of bugs



Calculation of time/effort investment in regular feature testing

Total person days invested in feature testing = Number of testers * product cycle (number of months) * time spent on feature testing

= 7 * 15 * 22 days * 0.7

= ~1600 person days

Assumptions:

- 1) A tester is spending 70% of his/her time in writing and running test cases, remaining 30% is being invested in handling bugs.
- 2) The testing team is of 7 testers.
- 3) The product cycle is of 15 months.

Comparative analysis: Prerelease beta test program effort vs. regular feature test effort

S. No.	Areas	On Prerelease/Beta test	On Regular Feature test
A.	Number of days spent by the tester	225	1600
B.	Number of bugs detected	387	1387
C.	Number of valid bugs (Fixed + Deferred)	63	1109
D.	Number of bugs closed as Fixed	41	922
E.	Average ROI per fixed bug (D/A)	0.18	0.57
F.	Bug success rate (D/B)	0.11	0.66

Findings:

As per the data, it was found that during a *regular feature test cycle*:

- 1 *valid (Fixed + Deferred) bug* is detected after every 1.1 days of QE effort.
- 1 *product improvement (Fixed) bug* is detected after every 1.73 days of QE effort.

On the other hand, it was found that during a *beta test cycle*:

- 1 *valid (Fixed + Deferred) bug* is detected after every 3.57 days of QE effort.
- 1 *product improvement (Fixed) bug* is detected after every 5.48 days of QE effort.

Beta testing is a strong platform which sometimes brings in critical bugs which were not detected during the months of regular feature testing.

This data helps the test manager to re-evaluate the process that is being followed while doing the beta testing. She should look out for areas to optimize in order to get maximum ROI from each QE resource being invested to the product and project.

Some of the prioritization areas that she can consider for improving the beta test phase ROI are:

- Optimizing the beta test program frequency. In the case discussed here, if the test manager reduces the number of beta releases by 1, she might be able to save an average of 30%* of QE effort. She can re-route the saved QE effort to other critical activities like regular feature test cycle.
- Defining a good communication method for beta testers. The above data shows 33%* count of the Closed or withdrawn bugs are of duplicate nature. This might reflect a case where the testers are not coordinating or communicating with each another before reporting bugs. Many of the beta testers are logging the same failure as different bugs and therefore increasing the triaging effort of the product QEs. The test manager can now design a

mechanism for logging a single bug, allowing other testers to add their votes to the same bug. This way, the bug count will remain one, QE effort will remain as a single unit, and all communication regarding the bug will be done through a single channel.

**Direct mapping of the cost saving is not an ideal case because each bug differs in its criticality and severity. The above data mapping is done to suggest a need for process improvement indicating a save and optimization of resources and time.*

A quick look at the bug severity distribution for regular feature testing bugs and Beta release reported bugs are as follows

With these strategies in place, a test manager achieves much higher ROI on the QE effort during the complete product cycle. Another way of looking at the benefits is that this strategy ensures the success of both the beta program and the regular testing cycle while keeping a perfect balance of organizational cost. While beta testing plays a significant role in a product life cycle and especially in cases where internal bug count slows down, it is extremely necessary to plan out the testing in all effective means to gain benefits at all levels across bugs, features getting tested, resources that are being used etc.

References:

1. *Dimension of Productivity concept: Metrics and Models in Software Quality Engineering, IInd Edition, Stephen Kan.*

A Distributed Requirements Collaboration Process

Brandon.Rydell@gmail.com
Sean.D.Eby@gmail.com
Carl.Seaton@gmail.com

Abstract

Distributed project teams struggle to accurately document the negotiations and trade-offs required to lead teams to concise, prioritized requirements. The main reasons for this challenge are 1) a lack of objective criteria to evaluate requirements and 2) a lack of freely available tools to channel negotiations into objectively characterized requirements which can be comparatively evaluated.

While there are numerous publications and tools addressing the challenges of software requirements engineering, many organizations today still manage requirements as tabular lists. Individual requirements are often modified during negotiations between groups such as engineering, marketing and management. Frequently much of this negotiation takes place informally in numerous face to face meetings where much of the rationale for changes and trade-offs are not documented and are subsequently lost.

In this paper we outline a new requirements collaboration process to address these issues. This new process is based on work done by Karl Wiegers in his paper [Prioritizing Requirements](#) [1] and is extended to include a method of proposing alternate requirements, documenting the negotiation of priority among interested parties and a way to rationally select priority requirements based on an objective measure of their relative merit. We also introduce an application prototype called the Distributed Requirements Collaboration Tool (DRCT) that was built for use by distributed teams to support the requirements negotiation process described above and also to address the issue of capturing rationale as requirements are negotiated. Lastly, we will discuss our experience using the DRCT to identify priority requirements for a more functional version of the DRCT.

Biography

Brandon Rydell is a software engineering supervisor at Portland General Electric (PGE) in Portland, Oregon. His experience over the last two decades includes working as a software requirements engineer and project manager (PM) on a wide variety of software engineering projects for PGE, PacifiCorp, Nike and United Parcel Service. Brandon has a Masters of Software Engineering (MSE) degree from Portland State University, and a B.A. in Business Administration from the University of Washington. He is also a certified Project Management Professional (PMP).

Sean Eby is currently a software engineer at PGE in Portland, Oregon. In his 11+ years of experience he has worked in all areas of software engineering for companies such as: Coaxis, Emery Worldwide, Merant and NCD. Sean specializes in designing and developing simple, elegant software solutions. Sean has an MSE degree from Portland State University.

Carl Seaton is a Staff Systems Software Engineer at Arris, currently developing high-performance video-on-demand servers at the Beaverton, Oregon site. Over the past 16 years Carl has done everything from customer support to software development to engineering management, specializing in highly reliable distributed systems. Carl holds B.S degrees in Computer Engineering and Computer Science from Oregon State University, and will complete his Master of Software Engineering degree at Portland State University this Fall.

Copyright: Brandon Rydell, Sean Eby and Carl Seaton August 1st 2009

Published at the Pacific Northwest Software Quality Conference 2009

Introduction

Fred Brooks said, "The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later." [2]

Requirements elicitation, specification and management are the fundamental tools used by software engineers when deciding precisely what to build. A common challenge with requirements management for organizations is to accurately document the negotiations and trade-offs that are required to lead teams to concise prioritized requirements, especially within distributed teams.

Two main reasons for this challenge include:

1. A lack of objective criteria to evaluate requirements (frequently requirements are dictated by vertical teams like marketing / engineering, etc.. and presented as simple lists) and
2. A lack of freely available tools to channel negotiations into objectively characterized requirements which can be comparatively evaluated.

While there are countless books, papers and websites addressing the challenges of software requirements engineering, and there are even commercial requirements management tools available, many organizations today still use basic documents or spreadsheets to manage simple requirements lists. This is due to a lack of software requirements engineering knowledge, skill and application.

In some more advanced organizations, requirements are often modified during negotiations between engineering (technical feasibility and cost), marketing (market benefit), and management (schedule, resources and strategic business goals). Frequently much of this negotiation takes place informally face to face during meetings where much of the rationale for alternatives, changes and trade-offs are not recorded. This challenge becomes increasingly difficult with physically and / or temporally distributed teams.

In this paper we outline a new requirements collaboration process which addresses these issues and as a result offers a process that is more suited to the needs of distributed teams. The development of this new process began by using Karl Wiegers' requirements prioritization process as a base. We then extended it to include a method of proposing alternate requirements, negotiating the priority of requirements among interested parties, capturing the trade-offs made along the way and rationally selecting priority requirements based on their relative rating on item (ROI) scores. See the Wiegers [Prioritizing Requirements](#) paper for a more complete description of a semi-quantitative approach to requirements prioritization, which we then adapted to develop our ROI concept.

We also introduce an application prototype called the Distributed Requirements Collaboration Tool (DRCT) built for use by distributed teams which supports the requirements negotiation process described above and addresses the issue of capturing rationale as requirements are negotiated.

Distributed Requirements Collaboration

We believe requirements elicitation, specification and management as a software engineering topic is of particular value to study for two main reasons:

1. The implications of Brooks' compelling statement above, made so many years ago, which in our experience holds true still today.
2. In our experience and training, the application of the discipline of software requirements engineering has been shown to have a good return on investment. Research by Boehm confirms this [3], yet software requirements engineering today remains insufficiently applied.

I. New Distributed Requirements Collaboration Process

When deciding whether it was necessary to develop a new distributed requirement collaboration process to meet the needs of distributed teams and to address the challenges given above we first reviewed two papers on the topic.

In Prioritizing Requirements Karl Wiegers describes a relatively simple process of rating and ranking requirements that is effective enough in our view, practical, and easy to use but does not explicitly extend well to distributed teams nor does it include support for capturing the rationale for trade-offs that frequently occur as requirements are negotiated and prioritized.

The paper titled Supporting Distributed Collaborative Prioritization for WinWin Requirements Capture and Negotiation [4] discusses a tool that was developed to support collaborating teams with prioritizing requirements in a distributed setting using the WinWin development model developed by B. Boehm. The tool includes support for multiple stakeholders, capturing rationale, and gives a much richer way to visualize the complexity of requirements prioritization. After review, our team decided that the resulting tool and process was more complex than we needed.

As a result, we developed the process summarized in the outline below (and also illustrated in Figure 1) which suits our specific needs. It is a simple process like the one introduced by Wiegers but extended to distributed teams and includes support for capturing rationale for trade-offs when negotiating the priority of requirements that are typical to projects we encounter.

Process Outline:

A. Propose a list of requirements that need prioritization. Note that requirements that are non-negotiable and must be included in a given project need not be prioritized and can simply be assumed part of the project.

For each proposed requirement:

1. Provide a name, description and rationale.
2. Assign (person in marketing role) a benefit and a penalty rating (based on a scale of 1 – 9, where 1 indicates low benefit or low penalty avoidance and 9 equals high benefit or high penalty avoidance) including rationale.
3. Assign (person in engineering role) a cost and risk rating (based on a scale of 1 – 9, where 1 indicates low cost or low risk and 9 equals high cost or high risk) including rationale.
4. Calculate an ROI. $ROI = (benefit + penalty) \div (risk + cost)$. The system performs this calculation based on ratings given.
5. Optionally, propose an alternate requirement and repeat steps 2 – 5 above.

B. Select the final list of requirements from the proposals negotiated above for inclusion in the cut list (person in the PM role).

Process Flow:

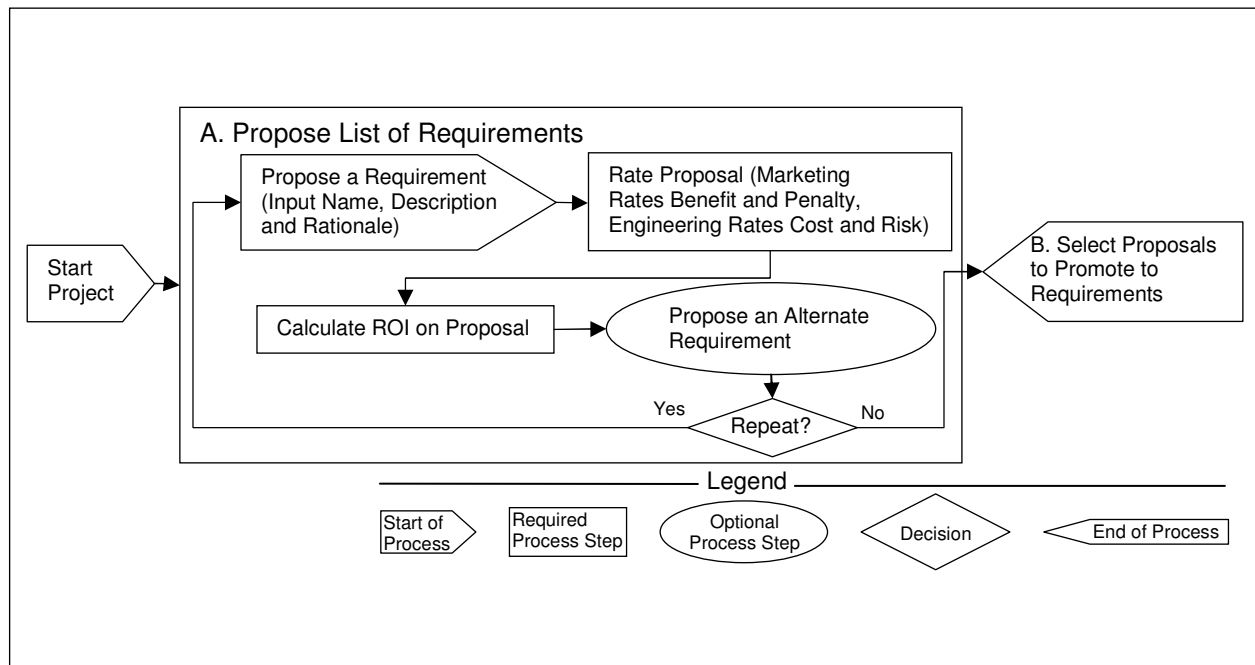


Figure 1 – Requirements Proposal and Negotiation Process

This process is effectively distributed by extending it to all project team members via an internet or intranet page (sample pages are provided in the following section).

From the process above, requirements are proposed, alternatives may be included, priority is effectively negotiated through ratings given by the respective roles, and because it is an application, a history of events including rationale is easily captured for the sake of documentation.

II. Distributed Requirements Collaboration Tool (Application Prototype)

The prototype we developed to support the process above is designed to perform the following functions:

1. Authenticate User
2. View Project(s) - projects are used to organize proposed requirements
3. Create Project(s)
4. Create Proposal(s)
5. View / Rate Proposal(s)
6. Propose Alternative(s) - alternatives are simply another independent proposal
7. Promote Proposal(s) to Requirement(s)
8. View Requirements

Note that in steps 4 – 6, the rationale for each can be recorded in the DRCT.

The sample wireframe screens below are taken from the actual prototype we developed and deployed to the internet. These screens are provided below in the same order as the processes listed above. Both the Authenticate User and Create Project(s) steps are not illustrated due to their simplicity and to preserve brevity.

Note that we used the colors and symbols below to indicate the following in the prototype:

- Grey and ■ - Current user's role (PM, marketing or engineering) does not allow editing.
- Red and ► - Rating is incomplete and needs to be filled in by another role or in the case of the ROI field it means that one or more of the ratings required by the current role needs to be completed before the system can calculate ROI.
- Green and ▼ - Rating is incomplete and needs to be filled in by current user, or in the case of the ROI field, it means that all the permissible inputs for the role currently logged in are complete.
- White and ● - Rating is complete.

1. View Project(s) – Upon logging in, the user will be taken to the “Projects” page. All projects are viewable here. No projects will be shown if none have been created. Note that multiple projects can be created by clicking the “New project” button shown at the bottom right. In this example we assume that a PM views or creates projects.

Distributed Requirements Collaboration Tool	
Welcome brandon!	Role: Project Manager Logout brandon
Projects:	
Name	Description
DRCT 1.0 Requirements	Project to collect and prioritize the non-basic requirements for the Distributed Requirements Collaboration Tool version 1.0.
Proto Testing	Proto Testing
Text Editor	Text
<input type="button" value="New project"/>	

2. Once a project has been created, it can be edited. More importantly, new proposals can be added to it. New proposals are added by clicking the “New proposal” button at the bottom right of this screen. In this example we assume that the PM creates a new proposal (shown next).

Distributed Requirements Collaboration Tool		
Role: Project Manager Logout brandon		
Editing Project: "Text Editor"		
Name:	<input type="text" value="Text Editor"/>	
Description:	<input type="text" value="Text"/>	
<input type="button" value="Cancel"/> <input type="button" value="Save"/>		
Requirements:		
Name	Description	ROI
Proposals:		
Name	Description	ROI
<input type="button" value="New proposal"/>		

3. Create Proposal(s) – This screen is used to create new proposed requirements.

Distributed Requirements Collaboration Tool	
Role: Project Manager Logout brandon	
Creating Proposal	
Name:	<input type="text" value="Unlimited Undo"/>
Description:	<input type="text" value="Undo all actions"/>
Rationale:	<input type="text" value="No one else does this"/>
$\left(\text{Benefit: } \square + \text{Penalty: } \square \right) / \left(\text{Cost: } \square + \text{Risk: } \square \right) = \text{ROI: } \square$	
<input type="button" value="Cancel"/> <input type="button" value="Save"/>	

3a. Once saved, new proposals appear on the “Editing Project” screen under the “Proposals.”

Distributed Requirements Collaboration Tool		
Proposal was successfully created.		
Role: Project Manager Logout brandon		
Editing Project: "Text Editor"		
Name:	<input type="text" value="Text Editor"/>	
Description:	<input type="text" value="Text"/>	
<input type="button" value="Cancel"/> <input type="button" value="Save"/>		
Requirements:		
Name	Description	ROI
Proposals:		
Name	Description	ROI
<u>Unlimited Undo</u>	Undo all actions	▶ <input type="button" value="Destroy"/>
<input type="button" value="New proposal"/>		

4. Proposals may be viewed and rated by selecting them by name from the “Editing Project” screen (shown previously). Here we see that an engineer has rated the “Cost” and “Risk” of the proposal.

Distributed Requirements Collaboration Tool

Proposal was successfully updated.
Role: Engineering | [Logout sean](#)

Editing Proposal: "Unlimited Undo"

[Back to Proposals](#)

Name: Unlimited Undo

Description: Undo all actions

Rationale: No one else does this

$$\left(\text{Benefit: } \blacksquare + \text{Penalty: } \blacksquare \right) / \left(\text{Cost: } 9 + \text{Risk: } 9 \right) = \text{ROI: } \blacksquare$$

[Alternate](#) [Cancel](#) [Save](#)

Alternate Proposals:

Name	Description	ROI
Unlimited Undo	Undo all actions	▶

4a. Here, marketing has provided ratings for “Benefit” and “Penalty” on the same proposal.

Distributed Requirements Collaboration Tool

Proposal was successfully updated.
Role: Marketing | [Logout carl](#)

Editing Proposal: "Unlimited Undo"

[Back to Proposals](#)

Name: Unlimited Undo

Description: Undo all actions

Rationale: No one else does this

$$\left(\text{Benefit: } 9 + \text{Penalty: } 9 \right) / \left(\text{Cost: } 9 + \text{Risk: } 9 \right) = \text{ROI: } 1.0$$

[Alternate](#) [Cancel](#) [Save](#)

Alternate Proposals:

Name	Description	ROI
Unlimited Undo	Undo all actions	1.0
Single Undo	Undo last actions	▼

5. Propose Alternative(s) – The “Creating Alternate for Proposal” screen is accessed by clicking the “Alternate” button on the previously pictured screen. Here, engineering proposes an alternative to the original proposal, which they suggest (in the “Rationale” field), will be easier to implement.

Distributed Requirements Collaboration Tool	
Role: Engineering Logout sean	
Creating Alternate for Proposal: Unlimited Undo	
Name:	Single Undo
Description:	Undo last actions
Rationale:	Much easier than undoing all actions
$\left(\text{Benefit: } \uparrow + \text{Penalty: } \uparrow \right) / \left(\text{Cost: } \downarrow + \text{Risk: } \downarrow \right) = \text{ROI: } \uparrow$	
<input type="button" value="Cancel"/> <input type="button" value="Save"/>	

5a. Marketing shown here enters an alternate proposal of their own and inputs their respective rating values and rationale as before.

Distributed Requirements Collaboration Tool	
Role: Marketing Logout carl	
Creating Alternate for Proposal: Single Undo	
Name:	Limited Undo
Description:	Undo last 5 actions
Rationale:	Easier than undoing all actions, more functional than only undoing the last action
$\left(\text{Benefit: } \downarrow + \text{Penalty: } \downarrow \right) / \left(\text{Cost: } \uparrow + \text{Risk: } \uparrow \right) = \text{ROI: } \downarrow$	
<input type="button" value="Cancel"/> <input type="button" value="Save"/>	

The negotiations continue until a list of mutually rated proposals and / or alternates is created. In the examples above, proposals and / or alternates once rated by marketing, will then be rated by engineering and vice versa.

6. The PM then, based on calculated ROI, evaluates each of the negotiated proposals and / or alternatives, and then selects those which he wants to promote by clicking the “Promote” button. In this example the PM is viewing the “Limited Undo” proposal just prior to promoting it.

Distributed Requirements Collaboration Tool														
Role: Project Manager Logout brandon														
Editing Proposal: "Limited Undo"														
Back to Proposals														
Name:	Limited Undo													
Description:	Undo last 5 actions													
Rationale:	Easier than undoing all actions, more functional than only undoing the last action													
$\left(\text{Benefit: } \boxed{7} + \text{Penalty: } \boxed{7} \right) / \left(\text{Cost: } \boxed{4} + \text{Risk: } \boxed{4} \right) = \text{ROI: } \boxed{1.8}$														
<input type="button" value="Alternate"/> <input type="button" value="Promote"/> <input type="button" value="Cancel"/> <input type="button" value="Save"/>														
Alternate Proposals: <table border="1" style="width: 100%; border-collapse: collapse;"> <thead> <tr> <th style="width: 25%;">Name</th> <th style="width: 50%;">Description</th> <th style="width: 25%;">ROI</th> </tr> </thead> <tbody> <tr> <td><u>Limited Undo</u></td> <td>Undo last 5 actions</td> <td>1.8</td> </tr> <tr> <td><u>Unlimited Undo</u></td> <td>Undo all actions</td> <td>1.0</td> </tr> <tr> <td><u>Single Undo</u></td> <td>Undo last actions</td> <td>0.5</td> </tr> </tbody> </table>			Name	Description	ROI	<u>Limited Undo</u>	Undo last 5 actions	1.8	<u>Unlimited Undo</u>	Undo all actions	1.0	<u>Single Undo</u>	Undo last actions	0.5
Name	Description	ROI												
<u>Limited Undo</u>	Undo last 5 actions	1.8												
<u>Unlimited Undo</u>	Undo all actions	1.0												
<u>Single Undo</u>	Undo last actions	0.5												

Note that the rationale provided during the negotiation process can also be used as a consideration by the PM when deciding which proposal / alternative(s) to promote.

7. View Requirements – Finally, any team member can see all the proposals that have been promoted to requirements, as shown here. Note the option to “Destroy” proposals is present for use by the PM.

Distributed Requirements Collaboration Tool

Proposal was successfully promoted.
Role: Project Manager | [Logout](#) [brandon](#)

Editing Project: "Text Editor"

Name:

Description:

Requirements:

Name	Description	ROI
Limited Undo	Undo last 5 actions	1.8

Proposals:

Name	Description	ROI
Unlimited Undo	Undo all actions	1.0 Destroy
Single Undo	Undo last actions	0.5 Destroy

This entire process is iterative and can produce lists of objectively prioritized requirements of unlimited length. As such lists are built within the DRCT by following this process, a comprehensive trail of documentation is produced as a valuable byproduct.

III. Evaluation of the DRC Process and DRCT

Here we describe our experiences with using the DRC process and tool, and present our conclusions on the effectiveness of using them.

Wiegiers' Process

Using the Wiegiers paper as a base to build our process off of worked well. Using subjective scoring made it easy to understand, while the objective ROI provided an effective tool to support decision making. This simple, effective approach made for a solid foundation to build the DRCT on.

One of the early challenges we faced was consistently applying rating values to each of the proposed requirements. While the rating scores are subjective individually, they must be consistent relative to each other in order to produce usable ROI scores. The DRCT left this challenge to the operators but some automated mechanism to support this would likely make sense as a new feature.

We found the Wiegiers recommendation from the paper, to keep the requirements evaluated at a high level, to be true. We initially put in very detailed requirements, but found that the volume of data generated made it more difficult to manage all the proposals together. To overcome this, we grouped the detailed proposals together into higher level proposals. This increased the effectiveness of the process.

Using the Prototype

Using the prototype to evaluate the process gave us good insight into what did and did not work well. Doing so also gave us good feedback on what needed to change about the DRCT in order to make it more usable, plus it helped us define more valid requirements in the process.

Because the DRCT as a prototype was immature, we found several areas that inhibited our use of it, and therefore, the evaluation of the new process. One case in particular was the lack of rationale per rating value. This made it more difficult to understand why another team member scored things in certain ways, and that lack of communication made it harder to respond with alternatives that addressed those unspoken issues. We used email at first to overcome this but found it to be somewhat ineffective due to lack of context.

Because we built the DRCT as part of our MSE practicum, we had to severely curtail the scope of the DRCT to stay on schedule. This gave us time to complete the DRCT and use it for the evaluation, but the resulting simplicity of the DRCT made it less effective to evaluate the new process and tool under real-world conditions. Again, due to this simplicity, we did not need to spend as much time during negotiations because of the relative simplicity of the proposed feature set, and the lack of more complex real-world tradeoffs limited the insight we could gain. However, the initial signs were encouraging.

The negotiation process was effective in promoting dialogue between participants. It forced stakeholders to communicate and clarify requirements, rationale, benefit and cost, ultimately leading to better decisions supporting the selection of final requirements for a cut list. We found that capturing those discussions to serve as documentation behind the decision-making process was especially useful.

The promotion step within this process was also effective. Explicit promotion was beneficial by having a third party (the PM) determine when negotiations would conclude. This helped to maintain the project's momentum. Having an independent role choose the requirements to promote also gave more objectivity to the process. While engineering and marketing debated, each had to keep in mind that a third party would make the choice based on the persuasiveness of the arguments.

One of the more interesting discoveries was in the ability for the PM to override the ROI calculation and simply choose to promote a proposal to a requirement for more subjective reasons. For example, an error-handling proposal had the lowest ROI value, but the PM thought, and we agreed, that it needed to make the cut list regardless of the ROI. The ability of the Project Manager to override the ROI is a useful correction mechanism.

As a result of the process and DRCT evaluation, we think the new distributed process that we practiced using the DRCT was practical and would be applicable to building commercial software professionally. We ended up with a working prototype that supported negotiating requirements among team members who are physically and temporally separated. From this prototype we developed an excellent understanding of what features the first release of the new tool should have in order to be a powerful and effective requirements elicitation and management tool. We achieved mutual agreement on our cut list (discussed below) as a reasonable and valid set of priority requirements for the next iteration of the DRCT.

We would be interested in seeing the DRCT and process used on a more complex project in order to validate them further. Our initial experience with them suggests that they would be useful on small to mid-sized projects where prioritizing requirements among distributed participants is needed.

DRCT 1.0 Requirements Cut List

As mentioned above, the authors used the DRCT to negotiate a proposed new set of requirements for a next version of the DRCT.

Once each proposal had a negotiated ROI, a cut list of requirements was selected by the project manager for the next release of the project. Using the calculated ROI values, rationale and some judgment, it was easy to select the requirements which made the most sense functionally, and which were agreed to have the greatest immediate value to the product. This process of selecting a short list was led by the project manager and offered a powerful check on negotiations that could have otherwise run on indefinitely.

A sample from this negotiated cut list is included below as Figure 2.

Name	Description	Rationale	Benefit	Penalty	Cost	Risk	ROI
Create users	Enable creation of user logins which can be assigned to individual stakeholder groups.	People will need to be able to use the tool, and having an admin modify the database for this would result in unnecessary delays.	4	9	2	2	3.3
Project audit trail	Should have an audit trail for projects, requirements and proposals (who, what, when, why).	Tracking changes allows users to more easily reevaluate things in the future. It also allows for easier tamper detection and can help correct user errors.	9	5	5	3	1.8
Provide rationale for each rating value	Rationale should be collected for each rating value, as well as for the proposal itself. Rationale should be updated when the rating value changes.	Rationales are necessary so that the viewers know the meaning behind the rating value given. This enhances discussions and decisions in the future as conditions change.	9	3	4	3	1.7
Encrypted passwords	Passwords should not be sent or stored in the clear.	This is an easy security hole to exploit and would not be acceptable to many organizations.	1	7	4	3	1.1
Error handling + logging	Display friendly error to user, log error details in database.	Users should be notified of an error occurring, but the tool should also log the error to assist technical support and engineering.	4	7	6	4	1.0

Figure 2 – DRCT V 1.0 Negotiated Cut List

Conclusion

While accurately documenting the negotiations and trade-offs required to lead teams to concise, prioritized requirements is often a challenge for distributed project teams, it can be achieved with a process and tool fit for the job. Making this process and tool fit to the task of managing requirement negotiations is worth undertaking, as it will invariably save time and money on most software projects.

In this paper we have introduced a new simple requirements collaboration process and tool for use by distributed teams that we believe will make managing requirements more effective for them. This new process and tool includes a method and means of proposing requirements, negotiating alternatives, collaboratively rating the proposals to determine their relative priority and then rationally selecting a requirements cut list from all the proposals offered. Distributed teams using this process and tool will enjoy the benefit of not only completing this work remotely but also capturing the rationale for making some proposals into requirements as they negotiate each proposal's relative merit.

Our experience with using this new process and tool in a distributed setting was positive. We think this new process and tool would be useful on many of the software projects on which we typically work.

References

- [1] – Wiegers, K. 1999. First Things First: Prioritizing Requirements - <http://www.processimpact.com/articles/prioritizing.html>
- [2] – Brooks, F. 1986. No Silver Bullet, Essence and Accidents of Software Engineering
- [3] – Boehm, B. 1981. Software Engineering Economics
- [4] – Park, J-W. 1999. Supporting Distributed Collaborative Prioritization for WinWin Requirements Capture and Negotiation - <http://csse.usc.edu/csse/TECHRPTS/1999/usccse99-516/usccse99-516.pdf>

Acknowledgements

The authors would like to acknowledge and offer our gratitude to the faculty, staff and students at the Portland State University Oregon Masters of Software Engineering (OMSE) program, including Dr. Kalman (Kal) Toth, Dr. Stuart Faulk, professor Manny Gatlin and the rest of the instructors, staff and students supporting this program. Additionally, our thanks go to our wives and children, whose patience and support of us during the development of this project are truly appreciated. A final thank you goes to Carol Oliver of Stanford University and Ganesh Prabhala, who have offered their time and patience to review this work.

Developing Requirements for Legacy Systems: A Case Study

Bill Baker and Todd Gentry

Abstract

Many legacy systems were created without documented requirements. Over the years changes have been made, often without adequate documentation. Software quality suffers as the system becomes more and more complex. This paper describes a case study of bringing requirements management and other related process improvements to a software product, which has been successful for over twenty years.

Eight years ago, we realized that we needed drastic improvements in requirements. It became obvious that just documenting changes to the system by each development team was not adequate. We had cases where one team counteracted changes made by another team in a previous release, causing bugs to reappear.

The product utilizes rules based on compliance and regulation from multiple sources; federal, state, and local. The company offers a significant guarantee of compliance and meeting of the regulations. Quality problems represent a significant business liability.

A cross functional team of management undertook a project to significantly improve processes related to requirements. These included centralizing requirements using a newly purchased requirements management tool and developing guidelines for writing requirements. Much of this paper describes the guidelines that were found useful, as well as some failures. We will show how the implemented process improvements have eliminated the quality problems that were the original impetus for undertaking the project and have improved the overall quality of the product.

We will discuss the following, among other things:

- The benefits we found from having a centralized repository and how this helped us discover problems in our processes.
- How we organized requirements to fit with the existing application.
- Prioritization of requirements work and where to focus efforts.
- Style. We found that the style appropriate for capturing changes in legal language is different than what worked well to capture enhancements to software behavior and workflow.
- How we successfully propagated change to multiple functional teams.

Biography

Bill Baker is a software development manager at Sage in Beaverton, Oregon. He has been involved in software development, project management and process improvement for a number of years. While at Harland Financial Solutions, Bill led the improvements outlined in this paper.

Bill has a Ph.D. in Electrical Engineering from Washington State University and an excessive collection of other degrees from Washington State University and Michigan State University. Bill can be reached at bill.baker@sage.com.

Todd Gentry has been a software developer for 25 years. He is currently a Senior Manager in Product Development for Harland Financial Solutions, supervising a software development team working in the .NET framework upgrading the technology of a legacy system.

Todd is a Certified Scrum Master. He holds a B.S. in Computer Science and Mathematics from the University of Oregon. Todd can be reached at todd.gentry@harlandfs.com.

Introduction

This case study focuses on process improvements around a legacy application that generates loan documentation for all of the different locales in the U.S. Loan documentation consists of a pool of hundreds of forms. The details of the specific transaction determine which forms apply. The regulatory environment surrounding these forms is diverse and includes federal, state, and local laws and regulations. The location of the transaction determines the specific regulatory context that is applied when generating the language and formatting for the forms for the transaction. The application was first created just after HP introduced the laser printer as an alternative to pre-printed forms. As with many legacy applications there was no real specification for this system. We used the source code as the specification when defending legal challenges against the compliance warrantee that backed the software.

This study describes the challenges we faced and the changes we made while implementing a requirements management system. Several years prior to this, the development organization had addressed delivery problems by setting fixed release dates, reorganizing into a matrix style development organization and breaking work into manageable size projects as compared to one massive project for the whole release.

The paper starts with a description of the situation at the beginning of the study. This provides the context upon which the process improvements were made. In the next section we describe the implementation and discuss the challenges we faced. The challenges are described using a narrative style. Each section is concluded with a take away lesson that we believe can be applied by any organization in a similar situation. We conclude with a section describing areas that need improvement, or where we think we could have done a better job, and a summary.

Situation

This study starts in late 2002. The product we are working on is a Windows application that generates loan documentation using a proprietary rules based system. The text and format in the documents must be compliant with all applicable federal, state, and local banking laws and regulations. At that time, the development group used a matrix organization made up of the following silos:

- Product Management group
- Product Legal group
- Project Management group
- Requirements/Design group
- Software Engineering group
- QA group

Release dates were set, up to a year in advance. We had four releases a year that added enhancements, software upgrades and bug fixes. Two releases focused on major enhancements and two focused just on “compliance” to address minor changes in wording to keep documents in compliance with current laws and regulations. Work was organized into relatively small projects of 2 to 6 months. All projects that were completed by a specified cutoff date were merged, regression tested and released in the next release.

Product managers, in consultation with the product legal group, were responsible for defining the scope of each project. They attempted to group like work together into a single project. For example, a project might be formed around a specific enhancement. The scope would include the enhancement and fixes for bugs that had been reported in that area of functionality. While, in concept, the project was localized around a specific area, this did not always translate well into source code. One problem the development group had was multiple projects in the same release were touching the same area of source code. It was not always clear how to merge the code, resulting in quality problems.

Generally there were four to six projects active at any given time. At the beginning of each project, project managers formed a team from available resources consisting of someone from the requirements/design group, several developers and a QA person. The designer, working with product managers and the legal group, developed a specification for the project. The “requirements” document developed was really a change specification. Design documents were updated to further specify the changes to be made by software engineers.

There was no organized communication between project teams. Specifically, there was no central repository for requirements documentation. There was no mechanism to query what had been done in past projects in some particular area of functionality. Project documentation resided in a project specific folder on a shared file system. Because of this and because we were capturing changes rather than a more general description of desired behavior, we had quality problems which are illustrated in the following scenario.

There is a paragraph of legal language that is shared between the privacy disclosure and the promissory note. The Privacy project had the requirement to make the paragraph all caps. The product was released. Customer calls pointed out that the paragraph on the promissory note should be mixed case, not all caps. A subsequent project changed things to make the paragraph mixed case. The product was released. Customer calls complained that the paragraph on the privacy disclosure was now mixed case and not all caps. At this point a manager who had knowledge of both projects made the connection and corrected the problem so that the paragraph is all caps on the privacy disclosure and mixed case on the promissory note.

The teams generally focused on changes. Because of this narrow focus they did not often step back and see the bigger picture. Team members were aware from the design documentation and the source code that the paragraph was used by multiple documents, but they focused on the requested change, not thinking to ask if the change applied to all documents. This was compounded because of the lack of a central requirements repository where the designer might have seen previous requirements about the case of the paragraph. There was no way to look at a complete set of requirements for any particular design element where the conflict above might have easily been detected. We also did not have any tool support for traceability, so analyzing the impact of making a particular change was generally not carried out because of the difficulty of doing so. There was no way for anyone to assess impact other than to review source code.

The problem in this scenario did not result in breaking the compliance guarantees of our product, but it should be easy to see how something similar could happen that would result in a significant compliance problem.

At this point in time we saw that the solution to this problem was a centralized requirements repository and decided to implement this using a requirements management tool. We put out an RFP and evaluated the responses. We performed an in-house evaluation of the top two contenders which resulted in the final selection and purchase of a tool. The implementation of the tool is where this case study really begins.

Implementation

A fair amount of work needed to be done before we rolled out the new tool. An initial tool configuration had to be designed and processes and procedures developed. A committee of stakeholders was formed to oversee this process. We wanted to create a standard for all products in the group. Besides the product that is the focus of this case study, there are other loan and banking products that could potentially benefit. The product legal group has input to all products, and regulatory developments often impact more than one product. The stakeholders were management representatives from the different matrix disciplines as well as representatives from different product teams. We were aware that a successful implementation depended on buy in from all concerned and made an effort to organize a complete set of stakeholders.

Training and consulting was included in the purchase agreement. A three day implementation session was organized. The first part of the session was training for the stakeholders in the use of the tool. The second part was the initial organization of the data, based on our needs. This discussion was facilitated by the consultant. The implementation session was instrumental in getting buy in from all the stakeholders. The consulting services we purchased were a key to our success.

Much of the discussion during the facilitated implementation session focused on how to leverage the work of the product legal group for use in multiple products. Discussion also focused on different levels of requirements and traceability between them. We came up with a high-level structure that we thought would meet our needs, with the understanding that it would be refined and changed based on experience. This structure, shown in Fig. 1, represents different levels of requirements that we thought would be useful. We initially focused on the product requirements which are equivalent to what Wiegers (Wiegers 1999) calls Software Requirements Specification or SRS, because that is where we thought we would get the most immediate benefit. We left the development of the higher level requirements, on the left, to later.

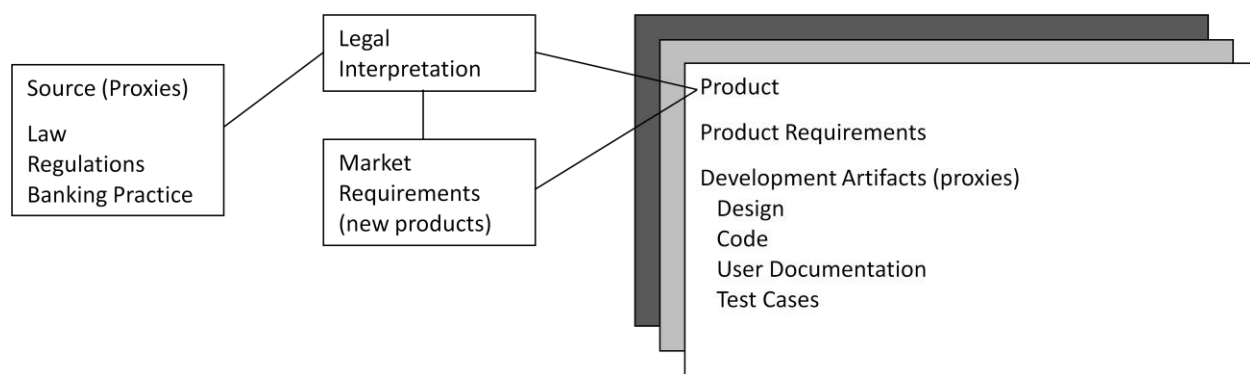


Figure 1. This diagram shows that we intended to have a single set of legal and market requirements that could be used by multiple products. We wanted the legal interpretation to link to laws and regulations.

Even the relatively simple relationships shown in this diagram weren't clear at the start of our discussions. Having someone from outside the organization facilitate discussions helped bring

out internalized knowledge that hadn't been explicitly documented until this point. This greatly helped in organizing our knowledge and was a key in a successful start to this project.

Lesson 1: Outside consulting services are key to a good start

We did the initial database configuration based on the structure that was developed. This was used by a pilot project. The main goal of the pilot project was to define and document how each discipline should use the new tool; in other words we were developing and documenting processes and procedures. The pilot was focused on the department's primary product with the assumption that the processes and procedures developed could be generalized to other product lines.

The first thing that the pilot team needed to do was to work out how the application should be represented in the requirements system. One of the main challenges was that there was no universal view of how the product should be represented. Our challenge was to develop consensus from all the interested parties. There were pros and cons to each alternative, but we did settle on one that was acceptable to all. Our top level structure was general and wasn't much different from what might be used for a brand new application. It is similar to the Software Requirements Specification (SRS) template by Wiegers (Wiegers 1999). The outline is listed below:

1. General Requirements
2. Definitions and Terms
3. Data Requirements
4. Data
5. User Interface Requirements
6. Product Features
7. Form Requirements
8. Documentation Requirements
9. Integration Requirements
10. Testing Requirements

Where things became interesting was when we started putting actual requirements into this structure. The people who were going to write the requirements, as well as the audience who was going to consume them, have deep experience with the existing product. In order for people to find information, the structure needed to easily map to their product knowledge. This differentiates legacy systems from new systems, where everyone is developing domain knowledge as they go and the product is still being defined.

The structure for the requirements that we developed reflects the structure of the product itself. The application builds documents or forms from paragraphs. Paragraphs are architectural elements in the application and can be reused in multiple forms. Rules are also reusable elements. A rule, for example, might be used to determine whether an element should appear in a particular location based on the transaction occurring in a particular geographic state. The same rule could be reused for anything dependent upon that state. We reflected this architecture in our structuring of requirements modules. Modules are high level elements in the tool we purchased and are equivalent to documents. The forms module listed forms and the paragraphs used in each by name. The paragraph names were linked to the section in the paragraphs module that

contained the requirements for that specific paragraph. We linked to rules defined in the rules module in a similar manner. This is illustrated in Fig. 2.

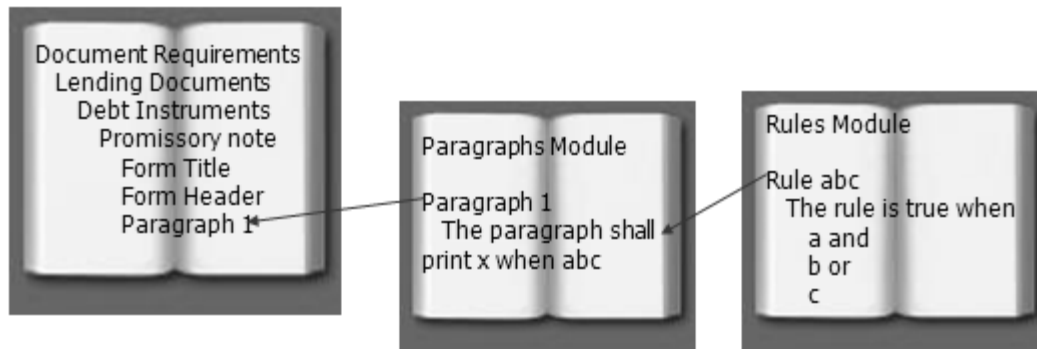


Figure 2. Diagram showing use of trace links to reflect architecture of application in requirements modules.

We made use of trace links in a unique way to model the relationships between architectural elements of the program familiar to everyone who worked with this product. The tool we purchased could display the requirements side-by-side which allowed the user to easily move from a higher level view to a detailed view. This is shown in Fig. 3.

7 Document Requirements	Links to Paragraphs	Links Paragraphs to Rules	Links to Rules
1.1.2.1.5 Iowa Consumer Notice Paragraph			
1.1.2.1.6 Vermont demand language			
1.1.2.1.7 Cosigner Notices Paragraph - Vermont Notice to Cosigner	<p>Paragraphs</p> <p>The notice shall print substantially as follows, once in each available Debt Instrument when the Vermont Cosigner Rule is TRUE.</p> <p>NOTICE TO COSIGNER: YOUR SIGNATURE ON THIS <NOTE WORD> MEANS THAT YOU ARE EQUALLY LIABLE FOR REPAYMENT OF THIS LOAN. IF THE BORROWER DOES NOT PAY, THE LENDER HAS A LEGAL RIGHT TO COLLECT FROM YOU.</p> <p>NOTE: This "notice" is meant to be appear COMPLETELY independent of any other type of cosigner notice.</p>	<p>Rules</p> <p>The Vermont Cosigner Rule shall be TRUE when all of the following conditions have been met:</p> <ul style="list-style-type: none"> • The <i>GSL State is Vermont</i> • EITHER of the following are TRUE: <ul style="list-style-type: none"> • The <i>All Borrowers Will Receive Money, Property or Services from Loan control</i> has NOT been selected and there is more than one <i>Individual Borrower</i> in the transaction. • The transaction contains at least one <i>Individual entity whose Capacity is Cosigner</i> 	
1.1.2.1.8 Fed Box for Disclosure Paragraph			
1.1.2.1.9 Lien Granted Pursuant to this Agreement Paragraph			

Figure 3. The left column shows the contents of the Document Requirements module. This module shows the structure of a document, in this case the promissory note, in terms of a list of paragraph names. The paragraph name is linked to a section in the Paragraphs module containing the requirements for the paragraph. The paragraph may link to rules in a similar fashion.

While the structure we developed for our existing application is probably not what we would have developed if we were creating a new product, the structure worked well because it was easily understood by the development staff, who were the primary consumers of this information. Another area where there is a direct mapping between the requirements structure

and the product is in the user interface requirements. The requirements hierarchy followed screens and workflow in the product itself.

Lesson 2: Product requirements reflecting product and software architecture make for easy initial use

There are many definitions of requirements. One definition (Sommerville and Swayer 1997) closely represents our working definition:

Requirements are...a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system.

We struggled with what should be in the requirements and what should be deferred to a design specification. Wiegers discusses this issue in general (Wiegers 2006). Our solution was to focus on how much we needed to constrain the rest of the development process. For example in some cases product legal wanted specific language under specified conditions. This did not leave the designer any leeway and we deemed this a requirement even though, to some, it seemed this was more appropriate for the design documentation. In other cases product legal would specify that a statement, such as a warning, was necessary and leave the wording up to the designer. In this case the requirement is for a warning, but not for specific wording. For this product, there seems to be a broad gray area between a requirement and design that is determined by the specific context.

This concept extended to the user interface as well. Certain features in the interface turned specific portions of language on or off and needed to be tightly specified in the requirements. In other areas the designer had more flexibility in what they could do. Since our requirements writers came with a background of tightly constraining things, one of the big problems for them was recognizing when this was not needed.

Lesson 3: Use requirements to constrain design only as much as is needed

Starting from no requirements documentation at all for a complex application is daunting. From the beginning there was a question of how much effort to put into filling in context around the requirements we needed to write for a specific element. Wiegers addresses this issue (Wiegers 2006). We strove to write the minimum number of requirements needed to understand what we wanted to do. This really is a question of how much context is needed. We wanted to provide enough context so issues such as the scenario cited above with the security paragraph wouldn't recur. This needed to be balanced against the business need of getting maintenance and new feature work completed.

We used several criteria to determine how much work to do. Bugs are often small and narrowly focused. We didn't see much benefit in creating requirements to capture changes needed to fix just bugs, and decided not to reflect these in our requirements. We did create trace links between any design and code changes and a bug tracking number to have complete traceability back to the reason for a change. On the other hand we found benefit in defining the context around larger changes or enhancements. Spending some effort on defining the context clarified the use case and resulted in a higher quality product because development had a better understanding of the desired product behavior.

Another criterion was the volatility of a particular area or document. We put in significant effort to capture current behavior as context for changes in volatile areas of documents with the anticipation that this work would pay off in the future by providing a base for future requirements work.

Lesson 4: Be pragmatic when filling in descriptions of current behavior

We structured the requirements so that they could easily be navigated by anyone with knowledge of the product. Developers are one of the main audiences for requirements and we addressed some of their needs by developing a style that effectively communicated what they needed to know. Much of the development work in any legacy application is maintenance. Developers are really looking for something that tells them what they need to change. This desire was particularly strong in our case because previous to this project, developers were given change specifications rather than requirements that described desired behavior. We developed a requirements style that communicated this change information.

To start we captured a complete set of requirements for the current product for a given feature. The version was labeled after these requirements had been reviewed and approved as accurately describing the current system. Next the requirements writer would modify the requirements to reflect the new desired behavior. We used a “change type” property to record the type of change to the requirements (modified, new, delete, or no change to existing behavior). After being reviewed and approved this version was also labeled. We developed some custom scripts that would compare the two labeled versions and create a redlined version that showed the differences. The redlined version and the change type designations proved very effective in communicating the desired changes to development staff.

The precursor to this process was the capture of a fairly complete description of the existing system. This provided a good contextual base for everyone. Modifying this base to describe the new desired behavior was effective in communicating the requirements to product management, the other main audience for our requirements. Having a complete description in a single location made review much easier than inferring behavior from partial requirements and other sources such as bug descriptions or the product itself.

Lesson 5: Find ways to effectively communicate to particular audiences

The product legal team has a big part to play in determining behavior for the product. But their training is in evaluating and writing legal documents, not in writing requirements. Requirements written in a standard style don’t provide the information in a form that is easy to review by this audience. Yet, because of the regulatory nature of the product we need legal input, especially for areas of some documents that depend upon a wide range of law and regulations. Writing requirements based on first principles would have had a high cost but would have provided little real business benefit. One technique we worked out was to write a small program that produced multiple versions of the same paragraph or document. This program iterated over the parameters that involved the legal or regulatory changes for a particular paragraph and printed a version for each different set of parameters. Pseudo code for this program is shown below.

```

FOR each value of parameter P1
  FOR each value of parameter P2
    ...
    FOR each value of parameter Pn
      Generate paragraph (P1, P2, ... Pn)
      Print paragraph
    END FOR
  ...
END FOR
END FOR

```

The number of unique paragraph versions could potentially range over a hundred depending on the number of parameters of interest and the number of values for each parameter. The output was passed to the legal team, who redlined the text. The redlined text became the requirements covering the particular changes in law or regulations for the paragraph. Although this is unconventional, it provided both an efficient and effective means of specifying language changes for the legal team, the developers who implemented the changes, and for the testers.

Lesson 6: Be creative in how requirements are generated and expressed.

We knew going into this that we needed to get everyone involved. The stakeholders group was formed with this idea in mind. We knew that without buy in by everyone in this group the implementation would not go well. As with any organization, there were organizational changes. In our case this occurred during our pilot. A new person became the manager for the requirement/design group, a key group in the adoption of the new tool. This person had not been part of the original stakeholder team. Specifically, this person had not been part of the tool evaluation and early planning. He was initially skeptical of the procedures we were developing. The pilot project timeline was extended so that we could address his issues and bring him onboard. The result was that we got buy in from this person, which greatly helped with implementation in the requirements writing team.

Lesson 7: Take time to bring people up to speed.

During the initial facilitated implementation sessions, we spent a fair amount of time defining traceability relationships between requirements modules and other development artifacts such as design and code. Creating the individual trace links fell to the development staff and became part of the updated development process. At the end of each project an audit is done to make sure that all requirements were completed in terms of updated design, source code, and tests. We also check to make sure that all source code changes can be traced back to a requirement or a bug report. This change in software development process has increased quality because we are now sure we have tested all changes. It also makes sure that we get everything we need to into release notes sent to customers. Prior to this project, developers would sometimes make undocumented changes, i.e. “fix bugs”, that they discovered during development. These changes didn’t always get properly documented and resulted in support calls from customers wondering about some unexplained change they had observed.

Lesson 8: Use traceability to improve software development processes

While we could have made a number of the changes without using a requirements management tool, purchasing and implementing a new tool did have one big benefit. It helped make change palatable to the staff. We’ve already discussed how we centralized and structured our

requirements. We had a lot of discussions around style and whether we should express something as a requirement or part of the design.

One of the biggest changes was moving from writing change specifications to expressing desired behavior. Not only did this improve our requirements writing for new features and enhancements, but it unlocked creativity from the whole team. It encouraged everyone to think about the best way to do something as compared to the mindset we had before that was narrowly focused on the change to be made.

Everyone recognized that they would have to change how they worked. This provided an opportunity to make more sweeping changes than we might otherwise have done without the advent of a new tool. Implementing traceability is one example. We will cite two more examples.

The first was an organization change away from a matrix structure to fixed teams. Fixed development teams were formed that were made up of several software engineers and a QA person. We also formed a separate release team that was dedicated to merging projects and producing final releases. One benefit was that team members developed better working relationships because they were together for much longer periods of time. The churn at the beginning of a new project was significantly reduced because team members already knew each other and had established working relationships.

A second example was a move to writing high-level test cases early in the development cycle, concurrent with the writing of requirements. This second view of desired behavior has become an effective means of communicating desired changes. It also removes the burden on the requirements of being the sole means of communicating the context of the changes.

The changes to processes and procedures we made as part of the implementation of the requirements management tool were significant. As a result, the organization as a whole became more accepting of change in general. One of the more significant changes that occurred well after the adoption of the requirements management system was a change to a more agile development method where we have a single backlog list from which the teams pull. This has resulted in greater development productivity. The adoption of Agile techniques was part of ongoing change that started with the implementation of the requirements management tool rather than as a separate initiative.

Lesson 9: Leverage tool adoption to make other changes

Areas Needing Improvement

While our requirements management and software engineering processes have improved greatly, there are areas that still need improvement. The first area we will focus on is impact and legal analysis. The second area is use by the legal team and teams for other product lines.

Going into this project one of the things we wanted to accomplish was the ability to easily do an impact analysis. We wanted the ability to quickly assess how big an impact a given change might require. We recognized that this required a critical mass of requirements which would take some time to amass. We have achieved only limited success with this. We have achieved a critical mass of requirements in volatile areas of the program, but these are fairly limited compared to the entire scope of the product.

When customers challenge our legal content, we have to research the legal reason for a particular piece of language or why a particular change was made. The people who do this still use the same tools as before; they look up the piece of language in the source code and use comments there to find related bug reports or call records. Traceability in the requirements management tool could be used to do this, but it hasn't been adopted. The ability to do this kind of research has improved because of the use of traceability implemented using the requirements management tool though. It is much more likely that code comments will include the appropriate reference ids now, than in the past, because of the use of trace links by the software engineers.

Lesson 10: Don't have high expectations for achieving impact analysis.

The pilot and initial implementation focused on the product requirements. A lot of effort was put into working out all the details of how different roles within the development organization would use the system. Initially we focused on the requirements/design team. The issues here were detailed organization, style, how much context to fill in, etc. Another big area had to do with the details of traceability and how to trace between various pieces of information. All this effort resulted in a system that was usable by the development group.

Much less effort was put into bringing the product and legal teams into the system. After the initial implementation effort, Bill, the lead architect, moved on to another project. This resulted in both of these teams not having full time requirements and tool support. They essentially had to fend for themselves in terms of figuring out how to do requirements in their group and how to best use the new tool. This was the case for other product lines as well. The result was that they continued to do things as they had in the past and never made the jump to effectively using the new tool or any possible improvements it might have provided.

Using the tool was particularly problematical for the legal team. The tool we selected represents requirements in a hierarchy much like you would in a regular written document. This didn't fit well with how the legal team worked with things. They work in three dimensions: legal concept such as indebtedness, legal form such as the note or collateral disclosure, and jurisdiction, such as a state. In a hierarchy, one of the dimensions is chosen as the top level or primary, and the other dimensions become sub-hierarchies under each section of the primary. The product requirements were structured with documents as the primary dimension. Jurisdiction is used to turn on or off specific paragraphs or other units of language. Given this organization it is almost impossible to determine whether some definition of indebtedness is applied consistently across the other dimensions.

We really have two problems here. One is that the tool does not support a multi-dimensional view where each dimension has equal weighting, or at a minimum can easily be queried with parameters from each dimension. The other issue is that the legal team really needs a different structure from what was used for the product requirements. For example, they would prefer to create a general definition for indebtedness, then describe jurisdictional level differences to the main definition and finally apply those to specific documents. This requires a mapping between the two structures.

Both of these are significant problems and would require time and effort to work through. Unfortunately a resource knowledgeable about requirements writing and use of the new tool was not available. The result was that the hoped for synergies between legal requirements and

product requirements expressed in figure 1 have not occurred. We suggest that a permanent position that can help teams with adoption and use would have been very useful in this case.

Adoption by several other legacy product lines was mostly unsuccessful for similar reasons. One new product did make successful use of it though. This was primarily because there was a member of that team who was experienced in developing requirements. The longer term implementation across the whole business unit would have benefited from someone who was experienced in requirements development and had a vision for the overall structure of all the requirements and how they should be represented in the tool, as well as detailed knowledge of the tool itself.

Lesson 11: Need “requirements architect” position

Summary

We have described the process improvements made during the adoption of a new requirements management tool. Specifically we have focused on the challenges of writing requirements for a legacy system that didn't previously have them. The lessons we have generalized from this experience are restated below:

- Outside consulting services are key to a good start
- Reflecting product and software architecture in the requirements makes for easy initial use.
- Use requirements to constrain design only as much as needed
- Be pragmatic when filling in descriptions of current behavior
- Find ways to effectively communicate to particular audiences
- Be creative in how requirements are generated and expressed
- Take time to bring new people up to speed
- Use traceability to improve software development processes
- Leverage tool adoption to make other changes
- Don't have high expectations for doing impact analysis
- Staff a “requirements architect” position

As we look back over the changes that have occurred we see that we are a much improved software development organization.

References

Karl Wiegers 1999, *Software Requirements*, Microsoft Press.

Karl Wiegers 2006, *More About Software Requirements: Thorny Issues and Practical Advice*, Microsoft Press.

Ian Sommerville and Pete Sawyer 1997, *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons.

Improve Quality by Making Clear Requests and Commitments and Avoiding the “I’ll Try” Trap

Pam Rechel
Brave Heart Consulting, Portland and Seattle
pam@braveheartconsulting.com
503 780-3965

Abstract:

Managers Slaying Dragons and Employees Being Thrown to the Wolves

It is widely known that the economic environment has contributed to difficult and stressful work environments. Often, employees are being asked to do more with less – fewer people to do the same amount of work, less budget and often greater client requests. And, the environment is changing at what seems like a faster and faster pace. The result is that managers are being asked to slay dragons every day – to solve complex problems that seem never-ending – and employees often think they’re being thrown to the wolves and getting eaten alive by all the work that comes their way.

Increase in Quality and Effectiveness

A question that this paper will consider is how making clear requests and commitments will result in great increases in effectiveness and quality of work especially in challenging times. Each time a fuzzy request is met by an unclear commitment, time and energy are lost. Considering how this is multiplied by many employees in an organization, it is easy to imagine how work slows down significantly while waiting for clear agreements about what work is really required and what can realistically be accomplished by when. This situation is slowed down even further when someone says “yes” when the answer to whether they can meet a request by a certain timeframe is really “no”. Unrealistic goals and unclear expectations are self-defeating for everyone.

Making Effective Requests and Commitments and Enemies of Saying “No”

The focus of this paper is to provide two methodologies; one methodology consists of skills to make an effective request and commitments. The other methodology is to understand the enemies and barriers that prevent us from saying “no” when we can’t meet a request and identifying some strategies to overcome those barriers. The skills of making effective requests and commitments include six practical steps. The steps are important, yet simple to apply. For example, it is much clearer to say “I need the updated Project Plan by the close of business on Friday” than to say “I need it as soon as you can”. This simple change can result in a more effective and efficient agreement.

All methods presented will be practical and could be applied the day after the conference.

Author Biography:

Pam Rechel, Principal of Brave Heart Consulting in Seattle and Portland, is an executive coach and organization consultant working with leaders, managers and their teams, especially in companies facing changing market conditions. Creating accountable organizations where it is easy to communicate and get results is a primary focus.

She has an M.A. degree in Coaching and Consulting in Organizations from the Leadership Institute of Seattle (LIOS), an M.B.A. in Information Systems from George Washington University, an M.S. from Syracuse University and additional coaching credentials from the Newfield Network in Boulder, Colorado. Pam holds the highest certification for Myers-Briggs Type Indicator® practitioners, including the MBTI® Step II. She facilitates workshops on Accountability for Teams. ©Brave Heart Consulting Inc. 2009

Introduction

What would your life be like if:

- Everyone you work with made clear requests and agreements?
- If you knew it was OK to say “no” when you knew you couldn’t meet a request?
- If you and others made commitments they would follow through on every time?

When **unclear requests** are made (for example, “Can you get me an update to the project schedule *as soon as possible*”) and are met with **unclear commitments** (e.g. “Sure, I’ll *try* to get that to you *when I have a chance*”), the end result is often frustration, extra time, low trust and most important of all, missed deadlines and poor quality because of lack of clarity about the request and the commitment when the request and commitment were made. Sometimes, the response is “I’ll try” when actually the person is aware that there is currently not enough time to get it done. But saying “no, I can’t get that done by the deadline” is seen as “not being a team player” or some other belief about why it’s not OK to say ‘no’. This lack of specific requests and commitments and the hesitance to say “no”, is experienced everywhere - in projects, work teams, families and the community. The result is that sloppy or vague requests often cause multiple ineffective conversations about one issue.

What will be described in this paper:

- Creating a culture of accountability
- What’s in it for you to make these changes in communication
- Six practical steps to clear requests
- Saying “yes” when you mean “no” - overcoming the “I’ll try” trap
- What are some enemies of or barriers to saying “no” and how to overcome them

Creating a Culture of Accountability

Making clear requests and commitments and making it acceptable to say “no” when a request cannot be fulfilled are some of the foundational skills for creating a **culture of accountability**. In a culture of accountability, the language that is used is clear and complete, typically in the first conversation. In a culture of accountability, commitments that cannot be fulfilled as promised are addressed directly and corrected. People say “yes” when they mean they can fulfill a commitment and say “no” when they cannot fulfill the request as it is initially made. In a culture of accountability, when someone says “no, I can’t meet the request as you requested it”, the next sentence is usually, “What changes can be made, such as taking things off my plate or changing the deadline, so I can meet the request?” In a culture of accountability, each person knows they have the skills and have the willingness to do what they personally can to improve a situation vs. saying that “someone else” should do something about this mess.

A culture of accountability applies to work organizations and teams, softball and other sport teams, as well as families and community organizations. This applies to all parts of life.

In this paper, the focus is on the language skills that provide part of the foundation of accountability: making clear requests and agreements and saying “no”. Resources for continuing the process of creating a culture of accountability are provided in the references at the end of this paper.

What's in it for you to make these changes in communication?

Let's go back to the questions "What would your life be like if..."

- People made clear requests and agreements?
- If you knew it was OK to say "no" when you knew you couldn't meet a request?
- If people made commitments they would follow through on every time?

Many people respond that their life would be easier and less stressful; they would get more things done quickly because they would know right away what they had to do and what others would really do. They would have a higher level of trust in the people they interact with because commitments would be fulfilled the first time more often and a conversation about missed commitments would be an easy process because it's expected and everyone would be trained in how to do it. "No" would stop being a difficult word to say to someone who made a request if you could not fulfill the request.

Benefits of a culture of accountability include:

- Builds confidence in making commitments or decisions
- Builds relationships and trust
- Prevents problems and saves time
- More efficient – reduces number of steps to getting things accomplished
- Saves frustration, waiting for someone to fulfill a request that was not clear or had fuzzy deadlines, quality standards or directions
- The skills can be applied immediately to many different situations
- Increased clarity to the job
- Builds competence

Six Practical Steps to Clear Requests

Why do we make requests? We make requests because we want something. We want cooperation, help, different results, someone to take action. It seems obvious that we make requests when we want to change the future by *asking* someone to do something. The first time I thought that a request changes the future, I realized that just *asking* someone to do something is taking action. I used to think that action was me having to do the work myself. Now, I see that the request is part of the work and making an effective request is part of how I can create the future and the results that I want. And I'm much more motivated to make clear requests because I can see how I'll benefit and so will the team.

Some of the steps to an effective request seem very obvious and simple, yet when analyzed, can be very challenging. For example, it may seem obvious that a Committed Speaker and Listener are requirements for an effective request. Yet, it is tempting for me to try to do two things at once and have a conversation to make a request while doing email or texting.

Sometimes we do not state clearly what future action and conditions of satisfaction are wanted because we don't know that they are. We may not have taken the time to analyze what's required and make a fuzzy request because we lack the clarity. In this instance, often the results are mediocre or not what is needed and have to be re-done. The cost is time for doing it over, frustration and lower quality. And, I don't have time, can't stand low quality and really don't have the energy to be unnecessarily frustrated!

Six Elements of Effective Requests

1. **Committed speaker – you**

It is not an anonymous or vague requestor. A committed requestor does what it takes to get a committed listener. You are not doing anything but making the request – not doing email, texting or talking on a cell phone. Example: An uncommitted speaker might toss a request over their shoulder as they pass someone in the hall. A committed speaker will stop and directly address the person. A committed speaker understands the objective and clearly frames and describes the request.

2. **Committed listener**

Not someone on the phone, doing emails, texting, talking to someone else or on their way out the door. That's right, neither you nor the person you're talking to should be reading email or texting at the same time you're making the request. Both of you need to stop and take the time, often one minute, to take the first action in making the request. Ask the listener "Is this a good time for us to talk about....If not, when would be a good time?" Sounds simple, yet sometimes it's not easy to stop and ask. Can you relate?

3. **Future Action and Conditions of Satisfaction**

Give enough detail about what will satisfy your request so that what is "obvious" to you is also "obvious" to the other person. For example, "What I mean by 'update the Project Plan' is to revise the timelines for all the action items for our team. I'd like the report to be sent to all the Project Leads". I know that you might not want to have to give all that detail. You might be concerned that the listener will think that you're talking down to them or telling them something they already know. And, until you tell them exactly what you need, without any assumptions that they already know it, you can't be sure they do know it. Besides, telling them is another chance for them to ask questions and to be sure you'll get what you need.

4. **Timeframe**

Detailed enough to get the results you want. Not "as soon as possible" or "when you have a chance" or "in a timely fashion" or "promptly" or by the end of the week (is Friday the end of the week? Is Saturday? Sunday?). This change alone was the best and easiest way I began to make more effective requests. I realized that I was fuzzy and not really telling others when I wanted something. Now, I'm forced to figure out when I really do want the request to be fulfilled before I make the request. Sometimes it's a 20 second thinking process, "Do I need this by Thursday or Friday?" and I can usually find 20 seconds!

For example, "It needs to be submitted electronically by the close of business on Friday (5 p.m.) Pacific Time"

5. **Mood of the request**

The frame of mind, yours and the other person's, affects how the request will be received. If you (or the other person) are angry when the request is made, the person listening will be less likely to respond favorably or be able to hear the request. Determine your mood and estimate the mood of the other and determine if it's suitable to make the request. Sometimes making a request when angry will work. But often the other person can't hear anything except the irritation. When the irritation/anger is all they hear, they can't hear your request. If you determine you or the other person is not in the right frame of mind, wait for a more appropriate time or change your frame of mind. (Of course, you can only change *your* frame of mind!)

6. **Context.**

Context is everything. Context is background information that can be helpful in preparing the listener to listen to what you have to say. For example, "The reason this request is important is that the main client has an opportunity to use our software with one of their new clients if we can guarantee the project completion date." In another context, say without a client request, the deadline not be important enough to implement.

Saying “yes” when you mean “no” – Enemies and Barriers to saying “no”

Why is “no” sometimes a difficult word to say? Let’s say the following request is made, “Can you revise our department budget this week?” If the organization culture is such that saying “no” is considered to be “not a team player”, then team members will say “yes” or “I’ll try” when realistically they have more commitments than are possible to fulfill in the required timeframe.

Enemies and Barriers to saying “no”

We each have some habitual reasons we might say “yes” when we mean “no”. There may be certain situations where this is more common – never saying “no” to your favorite professor, your spouse, parent, child or your best friend. An “enemy” is a common reason or habit that gets in our way of saying “no”. Review the following list of typical enemies and barriers to saying “no” and determine which ones are the favorites, or most commonly used, for you and/or your organization.

Common Enemies of Saying No

1. **Discomfort with saying ‘no, I can’t meet that request’.** Sometimes we grew up in families or now work on teams where we have to say ‘yes’ and saying ‘no’ isn’t allowed. Saying ‘no’ often does require personal courage and the belief that telling the truth clearly is the most respectful action that can be taken. *Saying “no” to a request or offer is not a rejecting the person, it is simply declining the request or offer.*
2. **Needing to be positive and seeing ‘no’ as a negative response.** This response is related to wanting to please and be liked. This calls for seeing ‘no’ as a truthful, respectful, clear response that is full of integrity. It is, in fact, a very positive response because you hold the requester as capable of hearing the truth, hold yourself as capable of telling the truth and believe that saying ‘no’ clearly will result in the most efficient response.
3. **Wanting to be a good team player.** We’ve commonly seen good team players as being ‘can do’ folks who will take on any task. Unfortunately, there are limits to what can be accomplished by anyone. Knowing your limits and when you can or can’t fulfill a request is an important part of being on a high performing team. High performing teams promise what they can deliver and deliver what they promise.
4. **“I can just work a little harder and try to get it done.”** This is a particularly seductive belief because it eats away at our ability to set limits and promise only what we can deliver. This devilish voice in our heads adds just a few minutes to every day until we’re always working. I’ve gotten sucked into this one more times that I’d like to admit. And, I’m getting better at realizing when I’m at the edge of what I can do realistically and that cutting out exercise or hiking so I can work a little harder and get a few more emails done doesn’t serve me or the organization.
5. **Living in constant over-commitment.** Always being overcommitted often becomes a habit and leads to burnout. It’s a primitive strategy that we learned early due to messages such as “everyone needs to give 120%”. The result is that you run out of full capacity, experience exhaustion and often find yourself in a mood of resignation (meaning you think there’s nothing that can be done about it...it’s just the way it is and will always be). Worst of all, you’re not producing quality work efficiently and often your non-work life and health suffer. Ask yourself if you’re living in a perpetual state of having way too much to do and commitments that you made not be completed. How is that working for you?

How to Say “No” in a Culture that Won’t Allow “No”

Sometimes organizational cultures simply won’t allow people to say “no”. I get that. So what’s a person to do when they can’t say no, but also know they can’t meet the request and they’re tired of making false commitments?

Here are some alternative responses that get to the same desirable end result: an agreement that can be fulfilled. Use these after you’ve exhausted all the other enemies of say ‘no’ and making sure that you’ve carefully examined whether you just don’t want to try out saying ‘no’.

Other Ways to Say “No” to the Request as Originally Made

- I can do that if the due date changes to...
- I can complete that if I take something else off my plate
- I’m already booked up between now and when you want this.
- I want to say ‘yes’, but only if we change the scope
- “I can if....”

Overcoming the “I’ll try” trap

What are the results when someone says “I’ll try” or “yes” when they mean “no”?

The results are that relationships and trust are damaged because an inaccurate response was given. Yes, we begin to say to ourselves and maybe to others, “I can’t count on Marie (names have been changed to protect the innocent!) because she doesn’t follow through on what she promises.

Another result is that the item can often be delayed while the requester is waiting for someone to “try”. In the original Star Wars movie, **Yoda says to Luke, “Luke, there is no try”**. There is no “try” because either you make a commitment to actually complete the task or you say “no”. There are obviously times when we do not completely fulfill our commitments, but not because we had a halfhearted “try”, it’s because of many other issues such as underestimating the time or effort required etc. When we say “I’ll try”, it often is intended for us to have an excuse for not meeting the commitment.

In reality, saying “no” when it is the straightforward truth is the most respectful response that can be given. It assumes that the listener is capable of handling the truth and that all parties will be better served to have the truth on the table as soon as possible so an alternative can be found that will allow the request to be satisfied.

Next Steps: Where to Start?

You know you’ve been moving fast and not as clear about requests and commitments. You’d like to make some changes but it seems overwhelming to all of a sudden start saying ‘no’ and turning down requests. Where do you start?

First, just notice what you do. Notice how many times you say something like, “I’ll get back to you ASAP”, without being specific. Once you notice your common patterns, add a clear date or deadline to your request or response. It’s that simple. Then, notice the results. What happens when you’re more specific with your requests or commitments (e.g. when you say, “get back to me by Friday at noon” instead of “ASAP”)?

Second, take a minute before you ask someone to do something to write down, or at least clarify in your head, the exact request that you want to make.

Third, talk with your manager and/or your team about whether or not you're meeting deadlines and getting the results you want. If the answer is that you're not getting the results, have a team conversation about whether requests and commitments are clear enough. Use the Six Elements of Effective Requests as your guide.

Conclusion:

In this paper, foundational skills for creating a culture of accountability were presented: How to make effective requests; understanding the barriers and enemies to saying "no"; and why saying "I'll try" is an ineffective response to a request were also discussed. Making these changes in your team, organization, volunteer group and/or family may lead to greater clarity, trust and getting more accomplished with less stress. Start today!

References and for more information:

Connors, Roger, Smith, Tom and Hickman, Craig. *The Oz Principle: Getting Results Through Individual and Organizational Accountability*. Prentice-Hall. 1994

Connors, Roger and Smith, Tom. *Journey to the Emerald City: Achieve a Competitive Edge by Creating a Culture of Accountability*. Prentice-Hall. 1999

Lundin, Stephen and Arnold, James. *Personal Accountability: Your Path to a Rewarding Work Life*. Charthouse International Publication. 1997

Miller, John G. *Flipping the Switch: Unleash the Power of Personal Accountability Using the QBQ!* G.P. Putnam Sons. 2006

Miller, John G. *QBQ! The Question Behind the Question: Practicing Personal Accountability at Work and in Life*. QBQ Press. 2004

Seiling, Jane Galloway. *The Meaning and Role of Organizational Advocacy: Responsibility and Accountability in the Workplace*. Quorum Books. 2001

Sull, Donald and Spinosa, Charles. Harvard Business Review, *Promise-Based Management: The Essence of Execution*. April 2007

The Elephant in the Room: Using Brain Science to Enhance Working Relationships

Sharon Buckmaster, Ph.D. and Diana Larsen
Published at the Pacific Northwest Software Quality Conference 2009

Abstract

Fusing data generated by current research in social neuroscience, positive psychology, and advanced imaging techniques, the brain science knowledge that results gives us new tools for understanding and enhancing the ability of men and women to work together. Taken together, this information is becoming known as gender intelligence and is being adopted by many progressive organizations. Companies like Deloitte & Touche, IBM, and PriceWaterHouse Coopers have seen immediate financial results including increased retention of women by training their managers to use gender intelligence in the workplace. Using the principles of brain science particularly with respect to gender differences can have a positive impact on corporate culture and organizational success.

Author Bios

Diana Larsen consults with leaders and teams to create work processes where innovation, inspiration, and imagination flourish. With more than fifteen years of experience working with technical professionals, Diana brings focus to the human side of organizations, teams and projects. She activates and strengthens her clients' proficiency in shaping an environment for productive teams and thriving in times of change. Diana discovers solutions and possibilities where others find only barriers and obstacles. Diana can be reached at dlarsen@futureworksconsulting.com.

Diana co-authored *Agile Retrospectives: Making Good Teams Great!*. Current chair of the Agile Alliance Board of Directors, she co-founded the "Agile Open Northwest" conference and the international "Retrospective Facilitators Gathering". She is a frequent presenter both nationally and internationally on Agile team development topics.

Sharon Buckmaster, Ph.D. coaches and consults with organization leaders wanting to create workplaces that are economically, ethically, and socially sustainable. She has developed a clear perspective on organizational change that builds on individual and organization strengths to meet current challenges. Sharon sees the art of leadership as a creative endeavor shaped by context and character.

Sharon excels at working with leaders. Her long-standing interest in leadership for women led her to found the Women's Center for Applied Leadership and her dissertation research called *Standing Up and Standing Proud: Senior Executive Women Who Advocate for Gender-Equity* (available at www.futureworksconsulting.com). She is an associate of the Center for Gender in Organizations at Simmons College and currently teaches in the Master's program for Applied Information Management at the University of Oregon. She can be reached at sbuckmaster@futureworksconsulting.com.

What's All the Fuss About Anyway?

It seems that everywhere we look these days, there are articles describing the implications of new research findings about the brain. From the pages of the N.Y. Times to Harvard Business Review to Fast Company, scientists are providing us with tantalizing new information that challenges much of what we thought we knew about what makes us tick and why we interact with others the ways we do. A few years ago, Dr. Eric Kandel, the 2000 Nobel Prize winner for biochemistry said in a speech that this research is clearly the new frontier in medicine and science, likely to vastly alter and also improve our understanding of individuals and society. We focus here on three of those research streams most relevant to people as members of organizational systems: social neuroscience, positive psychology, and imaging technology.

Briefly, we can describe these three areas in the following way:

Social neuroscience is the study of what happens in the brain when people interact. Daniel Goleman, perhaps best-known for his work on emotional intelligence, has developed an idea he calls “social intelligence “. Whereas emotional intelligence is primarily an intrapersonal process, social intelligence is an interpersonal one. According to Goleman, people are “wired to connect” with the result that we are inexorably drawn into an intimate brain-to-brain linkup whenever we engage with another person. That neural bridge lets us affect the brain—and therefore the body—of everyone we interact with, just as they do us. This neurological dance stimulates our nervous systems, affecting hormones, heart rate, circulation, breathing and the immune system. Goleman describes the relevant neural pathways, including the thalamus and amygdala, which together regulate sensory and arousal stimuli. He speaks of spindle cells, which rapidly process social decisions; of mirror neurons, which sense another’s movements; of dopamine neurons, which react to pleasure-inducing neurotransmitters that flow freely while two lovers gaze.

Positive psychology is a new branch of psychology focused on the study of well-being and the enrichment of human life. Researchers such as Dr. Jonathan Haidt at the University of Virginia have been studying the chemical and hormonal responses generated by witnessing acts of moral courage and inspired behavior. Dr. Barbara Fredrickson at the University of North Carolina has discovered that experiencing positive emotions in a 3-to-1 ratio with negative ones leads people to a tipping point beyond which they naturally become more resilient to adversity and can achieve what they once could only imagine.

Imaging technology is the world of functional MRI’s (fMRI), positron-emission tomography (PET) scans, SPECT scans, and other non-invasive brain-imaging technology that allows us to see inside the human brain in real time while it is solving a problem, experiencing emotion, or establishing trust with another person. Differences between men and women have been clearly documented in structural, chemical/hormonal, and functional areas. Researchers such as Daniel Amen, Michael Gurian and Barbara Amis have developed ideas commonly called “gender intelligence” to describe the likely impact of these differences on behavior.

Why Does “Gender Intelligence” Matter in Organizations?

Women have succeeded now at every organizational level and currently occupy just over 50% of the management and professional/tech roles according to 2007 data from Catalyst, a leading research and advocacy group for women in the workplace. Despite that progress, at current rates it will take more than 47 years for women to achieve parity in executive level, also called “C suite”, jobs. Clearly, the idea touted in the 1970s that simply having talented women “in the pipeline” would lead to gender equity, has proved inadequate. Numerous studies such as Harvard Business Review’s recent *The Athena Factor*, report that many women leave organization life because of hostile work environments and extreme job pressures. Ineffective communications, influenced in part by gender differences, can contribute to perceptions of discomfort that result in a decision to leave the organization. As retention rates fall, recruitment and retraining costs rise. After training its managers in brain-based gender differences, Deloitte & Touche saw such immediate improvement in retention rates that the company estimated it saved \$250 million dollars.

In addition, numerous academic and industry studies have documented high exit rates for women from the IT arena contributing to difficulties in filling roughly 500,000 information technology jobs nationally. Contrary to popular belief, there is now an ample supply of women graduating universities with degrees in the science, technology, and engineering (SET) fields. With more than 50% of the current US SET workforce approaching retirement age, organizations must examine strategies to address the workplace conditions that attract capable women and men, and increase the likelihood of their continued employment. Gender intelligence is one key tool that we can utilize.

So, how are our brains different?

There are three areas where significant differences exist between male and female brains. These are in actual structures of the brain, in the amount of neural blood flow, and in brain chemistry. Here are some of the key findings about the ways in which men and women’s brains differ:

- IQ tests of general intelligence show no overall difference...the difference shows up in different kinds of intelligence.
- Male brains show frequent rest periods i.e., “not thinking about anything”. This happens many times a day. Women’s brains do not do this except during sleep. Women often interpret this difference incorrectly as “withholding” behavior or a desire not to share.

- Male brains have 6.5X more gray matter. Gray matter serves as information processing centers. This localization drives focus, focus, focus. Females have 10x the amount of white matter, contributing to connectivity between the information centers, permitting more multi-tasking, more language facility, and faster emotional “processing”.
- Males have more M ganglion in the retina allowing men to perceive objects moving in space more easily. Women have more P ganglion allowing them to see color and fine detail.
- The hippocampus in men is less active, contributing to less linkage between memory and the emotional/word centers of the brain. Men will often not recall emotional discussions (positive or negative ones) whereas women will often have detailed recall.
- The amygdala in men tends to be larger. When angry, the verbal circuits in men tend to shut down with the amygdala driving towards more physical expression of the emotion rather than the verbal processing exhibited by females.
- Men have 20X more testosterone and also have vasopressin. Both decrease interest in talking and increase aggression, need for social power, competitiveness, and territoriality. These hormones were essential in an agrarian society for successful hunting. Testosterone rises and dips during the day, generally peaking at roughly 9-11 AM. Some research studies of British financial traders have shown that the riskiest trades tend to occur during that time period.
- Women have more oxytocin, sometimes called the “tend and befriend” hormone. Oxytocin promotes the personal bonding useful for building community and raising children, which are inherently social tasks performed over extended periods of time.
- In women’s brains, language tends to occur in both the right and left hemispheres, whereas in men language tends to occur only in the left. If you account for writing, speaking, and reading, women use significantly more words in a day than men.
- The cerebellum in men tends to be larger than in a female brain. The cerebellum is an action and physical movement center, contributing to the male’s tendency to action and physicality.
- Women have 20% more blood flow throughout the brain. In the limbic system, one of the consequences of this increased blood flow is that women are constantly assessing and reassessing context as well as facial expressions, tone of voice, etc. Disorders that inhibit people from picking up on social nuance such as autism and Asperger’s Syndrome are roughly 8X more common in males.

- Baby girls have as much estrogen in the brain as an adult woman. Estrogen is such a potent neurotransmitter that in the first three months of life, a baby girl increases eye contact and facial gazing skills 400% more than boys. This ability to read faces is an advantage females maintain over males throughout life. LouAnn Brizendine, author of *The Female Brain*, refers to the female brain as “a machine built for connection” in part due to this ability.

Do we have to be A or B?

Actually, one in seven men and one in five women have what we call “bridge brains”. These are people who have somewhat more of the other sex’s features in terms of their brain profile. A bridge brain man for example might take fewer physical risks than many of his peers, he might avoid a highly competitive type of profession, he might be drawn to less aggressive types of sports, etc.

Cognitive science tells us the brain is capable of change via focus or what David Rock and Jeffrey Schwartz in their work on the neuroscience of leadership call “attention density”. However, while the topic of plasticity in the brain has received a lot of attention recently, some aspects of brain functioning are not as amenable to change because they are hardwired. Nowhere is this truer than in gender-based brain differences. Nurture matters but it does not trump nature. It’s more useful to think in terms of the areas of the brain that have high versus low plasticity. An example of high plasticity would be the number of languages you can learn to speak. An example of low plasticity is the fact that women have more prolactin, which produces larger tear glands and in turn, more tears.

Organizational Implications

Once people understand some of the basic differences, they tend to look at everyday situations in organizations somewhat differently. Women (and some men) who use a lot of words may become more attuned to that “glazed-over” look from men they are speaking to who think they have already gotten the point. Men may begin to appreciate how women use their relational skills to build consensus and mend fences after controversy. Let’s say there is a very difficult meeting between two departments that becomes quite heated. The stress of conflict causes cortisol levels to rise, producing more testosterone in males and more aggressive, dominant behavior. In females, the rise in cortisol produces more oxytocin, leading to more harmony-seeking behavior. Perhaps the best outcome here would be a “winning” strategy where all the meeting participants feel good about the decision even if it took longer to get there.

Managers and coaches who understand some of the basic brain differences can adapt their style of interaction with members of the opposite sex as appropriate. Let’s say a female supervisor has a very angry staff person in her office. He is totally frustrated with his team. If the supervisor reacts the way a woman is more likely to react to another woman saying, “I can see how frustrated you are. Let’s go get some coffee and tell me how you’re feeling...”, we shouldn’t be surprised if the fellow gets really upset and even more frustrated. More useful for the man would be to offer some action steps like,

“Let’s make a list of the things you’ve already tried, then generate some alternatives you can implement right away”. Perhaps subsequently, you can pursue his “feelings” about the situation with the most effective approach likely to be asking a question such as, “What do you think is happening in the organization now that might be driving this behavior on the team?”

Another critical aspect of understanding brain differences is to understand that men and women tend to embody leadership somewhat differently. While at the very highest levels of organization life, the social intelligence needed for outstanding leadership shows no difference by gender, in the general population and lower organization levels there do appear to be some differences in the ways men and women behave in leadership roles. Recognizing these differences helps us to broaden, not narrow, our pool of prospective talent. Simply put, we can summarize these differences as:

Male leaders tend to:

- Bond in short bursts of connection
- Downplay emotion
- Focus on pattern thinking
- Promote risk-taking & independence

Female leaders tend to:

- Bond via extended conversations
- Display more “hands-on” connection
- Emphasize complex, multitasking activities
- Look for methods of direct empathy
- Be more willing to relinquish independence for “interdependence”

Summary

While biology isn’t destiny, we should not be afraid to acknowledge that there are influences on the brain that come from our gender as “males” or “females”. Admitting this does not have to lead to inequality or to inequity. Instead, we can use brain science to improve relationships and communication. Understanding these areas of difference without stereotyping can enhance the ways we manage conflict, negotiate, do sales, run meetings, and lead and coach teams. It can help us to build healthier environments where men and women can be authentically who they are and contribute their best.

Suggested Readings re: Brain Research

Brizendine, L. (2006). *The Female Brain*. New York: Broadway Books.

Goleman, D. & Boyatzis, R. (2008) Social Intelligence and the Biology of Leadership. *Harvard Business Review*, 9.

Goleman, D. (2006) *Social Intelligence: The New Science of Human Relationships*. New York: Bantam Books.

Gurian, M., & Annis, B. (2008). *Leadership and the Sexes: Using Gender Science to Create Success in Business*. San Francisco: Jossey-Bass.

Haidt, J. (2006). *The Happiness Hypothesis*. New York: Basic Books.

Hewitt, S., Luce, C.B., Servon, L., Sherbin, L., Shiller, P., Sosnovich, E., et al. (2008). The Athena Factor, Reversing the Brain Drain in Science, Engineering, and Technology. *Harvard Business Review Reports*.

Rock, D. & Schwartz, J. *The Neuroscience of Leadership*. (5/06) www.strategy-business.com.

Moving Software Quality Upstream: The Positive Impact of Lightweight Peer Code Review

Julian G. Ratcliffe

Senior Member of Technical Staff

Advanced Micro Devices

julian.ratcliffe@amd.com

Abstract

As schedules tighten and product launch deadlines remain fixed, software developers have to rise to the challenge without compromising quality. Managing software development and maintaining quality necessitates the development of processes and tools to facilitate efficiency. Software development teams, with a wide range of experience and expertise, are often spread across several continents. Coordinating an effort across such a workforce brings enormous challenges, not only because of the ever increasing complexity of silicon designs, but also the need to make excruciatingly efficient use of software development resources.

At Advanced Micro Devices (AMD) we have measurably improved software quality and built a culture of defect prevention by integrating a lightweight peer code review process into the development workflow. This paper describes the practical implementation and quantitative quality improvements. Additionally, the approach to process transition was effective in cultivating cultural acceptance, leading to developers adopting the changes almost voluntarily, something that surprised many of those involved!

Code review styles vary widely, ranging from “over the cube wall” to rigorous inspection, but many software development teams end up with a review process that is either chaotic or cumbersome. The addition of a lightweight peer code review process to coordinate and track the reviews introduces a powerful new weapon into the battle against bugs. Further, by closely integrating the code review, revision control and bug tracking, it is possible to design a process that can help and not hinder development.

Biography

Julian G. Ratcliffe is a Senior Member of Technical Staff at AMD. He has a background in parallel computing and holds a BSc. Hons. in Electronic Engineering from the University of Southampton, United Kingdom. His software experience spans more than two decades and in that time he has dealt with all processors great and small. He moved to Portland, Oregon in 1996 intending to spend a couple of years “getting the American thing out of his system”, but forgot to leave.

Introduction

Imagine for a moment, a river of code bubbling up from its source within the consciousness of a development team on its journey to the ocean of customers. Bugs* in the code are present as pollution and the art of clean programming is the effort of maintaining the clarity of the water. Impurities are inevitably introduced and the process of removing the contaminants becomes more expensive and time consuming as the river flows increasingly voluminously towards the ocean. It is therefore imperative to maintain the quality of the river of code by filtering it through a code review process as far upstream as possible.

Mention code review to most software development teams and the first thing that springs to mind are multiple meetings, reams of printouts and a lot of time that could be better spent coding. While a very rigid Fagan code inspection process may have been appropriate in the mid-70s, a significant amount of time and effort is required to collate review material and coordinate its distribution and review.^[1] Even less formal methods with less overhead, such as over-the-shoulder reviews or pair-programming still rely on authors and reviewers being able to meet, either in the same location or using teleconferences. But today development teams are not only distributed across multiple locations but also multiple time zones, so reviews can often only be performed asynchronously. A new better process is needed to efficiently conduct code review.

Examining the pros and cons of many traditional and modern code review techniques highlights a number of key factors that describe what the ideal process might look like. In order to be acceptable to both developers and reviewers a code review process needs to be easy to use and implement. It needs to make efficient use of their valuable time and not require explicit scheduling for either individuals or groups. Close integration with source code management and bug tracking systems is essential. Most importantly, from a quality perspective, the process should create a verifiable record of defect resolution and provide process enforcement. The process has to be effective at exposing defects and encouraging knowledge-transfer, naturally facilitating collaborative engagement across multiple sites and time-zones.

There are many factors to consider when modifying the workflow of a software development team, but the primary challenge is introducing change without disrupting existing commitments to schedules or deliverables. A productive team that is working to meet deadlines does not want to be distracted by process improvements that could be perceived as restrictive. Therefore the challenge is to introduce a lightweight process that does not impede current priorities.

The Lightweight Peer Code Review Tool

We chose the Code Collaborator code review tool from Smart Bear Software for our implementation.[†] The tool met our criteria for enabling the internationally distributed development team to perform code reviews in a repeatable and verifiable manner. Tasks such as gathering review material, notifying reviewers and coordinating review feedback are simple and intuitive.

* bug, n: An elusive creature living in a program that makes it incorrect. The activity of "debugging", or removing bugs from a program, ends when people get tired of doing it, not when the bugs are removed.

[†] A full description of the features or alternative tools is beyond the scope of this discussion.

The code review cycle starts with preparation for the review. Review material is attached with optional annotation and reviewers are assigned. Each review participant has a predefined role either as an author, reviewer or observer. Typically, the author allocates reviewers and observers from a list, but another mechanism allows an individual to use one of two subscription methods. One subscription method selects reviews by author for following changes made by a colleague. The other subscription method defines a file-grouping selection criteria used to watch changes in a specific part of the code tree.

Before the review begins the author has an opportunity to annotate the review material by adding notes describing the code changes. These notes are not a replacement for good use of comments within the code but are intended as a prologue for the reviewers examining the code. It's important to emphasize that this additional preparation represents a significant value to the reviewers in understanding the context of the review.

During the inspection and rework phases, the review is conducted using a web-based interface with an integrated chat-room style interface for comment and defect logging on a line by line basis. Conversations can occur in real-time or asynchronously. The customizable review templates define a set of roles, custom fields, and other options that determine the behavior and rules of a review.

The code review cycle is not complete until all of the reviewers are satisfied that the defects have been addressed and the review can close. The review is archived along with all of the associated activity and can be inspected or reopened at a later date. Metrics are gathered automatically as the reviews proceed and a configurable report generation capability provides a way to profile the data collected.

Integration with Issue Tracking and Version Control Tools

The introduction of a lightweight peer code review tool into the development process is only part of the solution. The integration of issue tracking and version control is essential in the successful execution of the overall code review process, in order to be tangibly effective in a large organization. Coordination with these other tools through their respective interfaces and hooks represents a significantly beneficial element of the implementation.

If you consider the three tools working together, the issue tracking tool defines the granularity of changes to the code base, the version control tool handles the change management and the code review tool introduces upstream quality inspection. Information from one tool can be used by the next, if at each stage of the process the continuity of the issue is maintained. By linking the tools we can verify the integrity of the entire workflow.

The issue tracking life-cycle begins when an issue is entered and assigned for assessment. This assessment may result in a task being generated and assigned to a developer, who in turn uses the version control tool to make code changes. The task forms the basic unit of work required to resolve the issue. The term "issue" is used not only to describe problems found in the code after the core development is complete but any work that is done on the code base at each stage of

development, debug, and maintenance. Using the issue tracking tool to manage the task granularity creates a verifiable record of the workflow that can be used to forecast future development effort. Each task remains active in the issue tracking tool until the code changes have been completed, reviewed and integrated into the code base, at which point the task can be closed.

Changes are commonly managed using task branching, which creates an explicit short term branch from the main trunk. This provides an isolated “sandbox” for the developer to work on any given task. The developer is then free to make as many iterative changes to the code as they require, testing the changes as they progress. The obvious and most convenient place to introduce the code review is just before the task branch is ready to be merged back into the main trunk.

By integrating the tools through their respective interfaces and creating a direct association with the task granularity, there are numerous opportunities to add automation and therefore reduce manual errors and tedious steps. A simple example is the use of hyperlinks, constructed using the task identifier, to quickly cross-reference information between each of the tools with a single click. Further examples might be the automatic attachment of review material from the version control system or the inclusion of the originator of a task from the issue tracker on a review.

An additional benefit we’ve been able to leverage from improved tool integration is a mechanism that more effectively generates release notes. In the past the release notes would be harvested from the issue tracking database and were often inconsistently completed. This was performed using a manual method immediately before the release was due and presented a significant challenge. Instead of entering the release notes into the issue tracking database a release note file is generated in the task branch and included in the review. Once the review is complete it can be pushed to the issue tracker interface using one of the configurable triggers provided by Code Collaborator. By tying the release note information to the review process for the code changes we were able to ensure that the review notes had been reviewed by several people and the details were entered while they were fresh in the minds of the developers.

The Importance of Checklists

When developers are reviewing code they will normally use a checklist to ensure standardization and identification of common errors. These lists can get very long as most are created to be comprehensive. Unfortunately, it soon becomes a tedious and unmanageable task when a couple of hundred lines of code are cross referenced against a list with twenty or more items. Even if the list is divided into categories, most people with average attention spans would not be able to maintain a focus on all of the items across all of the code. We need to keep the code review process sufficiently lightweight and efficient to ensure that we are gaining maximal benefit from a small number of reviewers.

There are a few questions that we need to answer when trying to develop a good checklist. How many items should be on the list? What should be, and should not be, on the list? How do we maintain the list over time?

Firstly, let's consider that research into the capacity of the human short-term memory shows we can handle about 7 items plus or minus 2 ^[2]. This would imply that the checklists should contain at most 9 items in order to keep them at the forefront of our mind as we scan through the code. Any more than that and we would need to repeatedly refer to the checklist in order to maintain awareness of any subset of items on the list, breaking our focus on the actual code under review.

As with debugging, the code review ends not when all of the defects have been discovered but when we become tired of looking for them. There is a significant risk of lowering the effectiveness of reviews if the checklists get too long and several passes over the code are needed to cover all the checklist items. If there isn't time to complete a full review be sure to at least look at the code with regard to a few points on the checklist. Even a partial review is better than no review at all.

Some review items don't appear on the checklist at all. An experienced developer has cultivated an instinctive set of personal rules for coding over the years, as they have learned from their own mistakes. These items do not need to be on the checklist since they can be considered as obvious to the individual reviewer. These "obvious" items, however, are communicated as each participant acknowledges the comments and defects marked by the other reviewers. This collaborative acknowledgment is an important aspect of the shared experience of reviewing the code. It builds not only a greater understanding and respect between remote teams but also provides the additional benefit of informally mentoring less experienced developers.

Most developers are looking for mistakes in the code itself, but just as important and often harder to spot, is what is missing. There are usually a number of things that are commonly forgotten. For instance, a developer is generally focused on how to make the code work and the reviewers will be looking to verify specific functionality. What is commonly missing is the consideration of error conditions. Well designed software will gracefully handle an unexpected execution path, so do not forget to add items to the review checklist that look for good error handling. The quality of the code is dependent upon things to go right even when something has gone wrong.

Additionally, remember that code review is not a replacement for testing and verification. There is little point keeping items on the checklist if they can be detected using an automated tool. The supplementary use of a static code analysis tool to check for stylistic and common programming problems is advisable. These tools can also reveal issues that are not easy to spot using a purely visual inspection of the code. A code review process typically ensures that developers run automated static analysis tests both before submitting the code for review and again after any rework is done.

Each code base and development team has their own dynamics which cannot easily be predicted. The code review tool allowed us to customize the defect reporting dialog and ensure that each defect found was identified by severity and category. By profiling the reviews to identify the most frequently tagged defects in each category, one can empirically build a prioritized list of commonly occurring issues. By using the frequency of the various categories of defects from the first few hundred reviews a checklist can be developed that is relevant to context of the development group. Once established, a complete checklist needs to be maintained by retiring items that have become stale and replacing them with those of a higher frequency or priority.

Building a good checklist is not as straightforward as grabbing an example from somewhere else. This data driven approach of eliminating the types of defects that appear to be most common in the code allows focus on those areas that require the most urgent quality improvement. Over time as the review metrics accumulate it is important to establish quantifiable goals towards process improvement and maintaining quality.

Which Metrics are Important?

The Code Collaborator tool collects information on the number of files and lines that have been modified, the number of defects found and the amount of time spent in the various phases of each review. Using these figures, configurable templates can be used to flag reviews that are considered trivial or stalled. For example, a review may be classified as trivial if no defects were found in a review of less than 30 seconds or stalled if no activity was logged for several days.

These raw numbers are influenced by many different parameters for a given group of developers or code base and will vary widely. While analysis may not provide a definitive measure of the effectiveness of the code review it is possible to interpret the data and identify rough trends to facilitate quality improvement.

Three measurements provide the basis for further analysis; defect density, inspection rate and defect rate:

- Defect density is the number of defects per 1000 lines of code. Considering that a reviewer may inspect files that vary in size from a few lines to a few thousand lines we would expect that more defects would be found in larger files. Likewise, the number of defects found in a review may differ given the wide variation in number of lines changed. By using defect density we can normalize the number of defects with respect to the amount of code under review.
- Inspection rate is the number of lines of code reviewed per hour of review and will typically vary from approximately 100 to 500 lines of code per hour.
- Defect rate is the number of defects found per hour of review. Again, reviews will vary in length depending on the number and range of the changes so it is useful to define these rates in order to normalize against time.

As we attempt to identify trends there are many factors that we need to consider. A large group will contain reviewers that are exceptionally meticulous but slow and those that are quick but more superficial. Inspection rates that are greater than 1000 lines per hour might indicate that reviewers were not inspecting code carefully but less than 10 lines per hour may be considered unproductive. Some files may be more sensitive to changes because they have algorithmically complex code or represent a reusable element that requires accurate implementation. Individual reviewers will be more or less effective at identifying defects than others based on their experience or familiarity with the code.

Even with some level of normalization we still need to be extremely careful when using these figures to draw conclusions. Trying to ascertain how fast the code should be reviewed or how many bugs per hour need to be found is not a useful goal in itself. Similarly, defect density does not reveal how buggy the code is. Abnormally low defect rates are not an indicator of high code quality and it is probable the opposite is true.

So can these metrics be meaningfully interpreted with respect to quality? As the number of reviews accumulates over time we expect to see a distribution evolving for each of the characteristics we are measuring. For instance, if we plot reviews measured by defect density we should see a distribution that we can use to determine our mean defect density. The same should be true for reviews plotted by inspection rate or defect rate. These means are important indicators which are useful for identifying the average standards of your group with normalization applied. The outlying data points may also warrant inspection though, since they may represent areas where a better understanding of the measurements is needed.

It is also important to perform periodic random audits on a small percentage of the reviews. The auditors should be an experienced team of reviewers looking for the highest level of compliance to expected quality standards. By comparing this sample against the overall statistics a determination can be made as to whether the mean values are acceptable or require action to be taken to improve them. This might be something like providing additional training, performing an in-depth review of a code tree or identifying a necessary process change. The metrics won't determine the absolute quality of the code but they will highlight issues that may require attention.

Gaining Cultural Acceptance

The introduction of a lightweight code review process is intended not only to move software quality upstream but also to create a collaborative culture of defect prevention. By creating an environment where discussion can conveniently occur not only around the water cooler but also around the world we essentially encourage an ethos of cooperative community. This may seem like an abstract concept, and difficult to measure, but it is hard to deny the benefits of creating a true sense of “team” given the realities of globally distributed developers.

So what can be done to encourage a group to enthusiastically adopt a new process? We took an approach based on consultation, prototyping, adoption and refinement. There was no “go live” date since the new infrastructure was developed alongside the existing one. The soft launch happened gradually and was expanded within the organization in stages. By presenting code review in a positive light and not as a punitive measure we aimed to convey a message of continuous improvement without highlighting an individual's productivity.

Initially we formed a small group of stakeholders in the code review process. Representatives from the software development, IT infrastructure and quality assurance organizations each had a chance to review the proposed process changes. Their feedback allowed us to tackle how the process could be designed to have a low barrier for adoption and acceptance, cause minimal disruption to the existing commitments on the schedule and scale as a wider roll-out.

As part of the prototyping a small team volunteered to be early adopters and help work out the kinks. We developed a small number of scripts to handle some of the most common operations such as the management of task branches and the attachment of review material to a review, eliminating many of the manual steps. The proof of concept was so successful that adoption spread initially by word-of-mouth and individuals began using the process voluntarily, even before they received training. We even reached some groups that we had not originally targeted, which led to further refinements in the process.

A number of concerns did surface during the prototyping that needed to be addressed. These related to ensuring that reviews did not stall and therefore impede code development. For example, one concern was how to shortcut the process when a delay to a release was not tolerable. The provision to allow developers to work around the new code review process was a useful part of the soft launch approach. The ability to skip the code review process and avoid using the task branches provided a safety net. Once the developers were familiar with the extra steps it became clear that the fear of being impeded by the process when a code change needed to be made urgently was unfounded. The process was lightweight and reliable enough to be workable at all times and ultimately the safety net was phased out.

A second concern was raised regarding how to allocate reviewers to reviews given an organization of over 100 developers. The allocation could stall the review if a particular reviewer was already overburdened, on vacation or simply inappropriately assigned. To overcome the allocation issue we implemented a concept of “reviewer pools”.

Reviewer pools are created in groups organized by technical expertise, geographical location or any other appropriate factor. The reviewer pools are implemented through an email group distribution alias. Individuals are able to subscribe or unsubscribe from the alias using an internal web page that also shows the current group memberships. When a review is initiated the reviewers can be allocated either as hand-picked individuals or opened up to one of the reviewer pools. This flexible allocation approach provides a mechanism to automatically broadcast notification of the review to the review pool subscribers.

Using this flexible allocation approach, individuals can then self-select to become a reviewer thus allowing a mechanism of cooperative load balancing. As long as the pool of reviewers for any given grouping is large enough, individuals can decide whether they are available to participate in any review to which they are invited. This structure allows people to manage their own review workload and also avoids delays associated with inviting an individual that may be legitimately unavailable. The problem of evenly distributing reviewers to reviews can therefore be alleviated.

Other teams external to the development group may choose to use a review pool alias acting as a brokerage instead of group distribution. Although the review pool is defined by technical area a notification recipient may act as a broker to reallocate the review role to an appropriate individual based on their expertise or availability. In either case, the reporting functionality of Code Collaborator is able to provide details of who has actually been allocated to the reviews and provide visibility of the workload balancing.

The consultative approach and prototyping stages were crucial in establishing credibility for challenging the status quo and introducing process changes. By demonstrating a flexible approach and incorporating process changes in response to feedback we cultivated a number of early advocates. Once an initial momentum had been established, adoption occurred at a steady rate as each group moved to align themselves with their peers.

Conclusions

We are currently at an early stage of adoption and still gathering a sufficiently large set of review data to perform a detailed analysis. Our early metrics show an apparently low defect rate of less than 1 defect per person-hour, with approximately only 1 in 10 reviews registering defects. A closer look at the review archive shows that reviewers were mostly engaged in discussion, using the comment threads to fix issues instead of logging defects. Issues fixed in the discussion threads are still valuable but they circumvent the metric gathering. The challenge is to educate reviewers to critically identify defects whilst not impeding the creative and collaborative process fostered by discussion.

The initial data regarding the average amount of time spent on reviews is more consistent with our expectations. The average total time per review is about 45 minutes with the average author time about 15 minutes and the average reviewer time about 30 minutes. These measurements are useful in gauging the cost-benefit associated with upstream defect prevention in comparison to late stage or post release fixes. With the introduction of a lightweight peer code review tool we have successfully managed to detect a significant number of defects that would have been more costly to eliminate later in the workflow.

It is important to consider that while the majority of code reviews involve inspection and feedback from a peer group of developers, there are several other cases where other parties, not involved in code development, can add value to the overall improvement of code quality. For example silicon design engineers have valuable input from the point of view of how the hardware was intended to be programmed and software support engineers have experience of system integration from a customer perspective. The code review tool allows us to easily connect these associated groups and facilitate discussion between them.

It seems apparent that additional training would help reviewers structure their approach to code review in a consistent and rigorous way. The inclusion of a checklist with the review material will remind reviewers of relevant items to look for and ensure inspection is performed in a systematic and objective manner. There are many possible factors related to software quality, such as conformance, reliability, maintainability, scalability, testability, documentation etc. The tracking of defects by category and severity will identify the most commonly occurring issues and help define and prioritize our checklists.

Successfully introducing a lightweight code review process is half the battle but we have a number of things we need to tackle next. The effective use of metrics will require rigorous analysis using standardized measurements to minimize the effect of individual reviewers or reviews. Once we can establish a baseline, the next step will be to set quantifiable goals and monitor our progress.

In conclusion, quality assurance is often a misleading term used to describe a process that would be more aptly named quality insurance, given that it occurs after code is developed. True quality assurance can only occur as development is in progress. To improve software quality an organization must be prepared to establish an objective measure and take a prioritized approach that is not compromised by schedule pressure. Implementing a modern, lightweight, and verifiable peer code review process represents a big step in that direction.

References

- [1] Fagan, M.E., *Design and Code inspections to reduce errors in program development*, IBM Systems Journal, Vol. 15, No 3 1976
- [2] Miller, G. A., *The magical number seven, plus or minus two: Some limits on our capacity for processing information*, Psychological Review, 63, 1956
- [3] Jason Cohen, *Best Kept Secrets of Peer Code Review*, Smart Bear Software

Acknowledgments

For their valuable feedback and review;

Miska Hiltunen, Qualiteers
Sara Gorsuch, Symetra Inc.
Suzanne Gillespie, Kaiser Permanente Northwest, Center for Health Research
Lisa Wells, Smart Bear Software

AMD is a trademark of Advanced Micro Devices, Inc.

Smart Bear® and Code Collaborator™ are trademarks or registered trademarks of Smart Bear Software.

Quality Cost Management

Manage Your Quality Costs or Let Them Manage You

Abstract

Many of us QA professionals spend our entire careers managing defects and we're getting pretty good at it. But if defect management is our universe, we are not really *assuring quality*. We need to move away from finding, reproducing, documenting, prioritizing defects and validating fixes. We need to move to preventing defects.

Rising above the defects takes commitment to process improvement. What is stopping you, dear reader, from making serious software process improvements? My bet is that your answer is "Time." If you are a software quality professional, the only other valid reason is "Knowledge." Gaining process improvement knowledge is easy – great resources exist ... if only you had time.

This paper addresses the time issue. The answer is money. That's it. It's that simple and it's that hard. Getting money to improve processes require a convincing business argument. Executives understand money more than they understand software quality. Your task, should you choose to accept it, is to convince your management that preventing problems is more profitable than finding and fixing them. Only then will they fund your improvement efforts.

Here's your starting point: Proactively managing quality costs is a sounder business practice than reacting to product failures.

This paper will provide the theoretical foundation for actively managing quality costs to increase profits and reduce chaos. My goal is to help you convince your executives to invest in defect avoidance... with dedicated resources driven by data. Quality cost management, based of Crosby's cost-of-quality foundation, means reducing overall quality costs by eliminating sources of waste and failure.

We need not swim in defects forever. There is a better way.

Biography



Ian Savage

A quality/productivity practitioner and evangelist, Ian is a veteran software developer, quality assurance engineer, manager and executive with experience in the high tech manufacturing, financial services, construction services, and security software domains.

Ian learned about quality at Tektronix and has served PNSQC in several capacities. He serves on the boards of the Software Association of Oregon, the SAO QASIG, the Tech Alliance: Central Oregon's Technical Association (SAO chapter), and PNSQC in various roles.

Trained as a process engineer, he has contributed to the Agile Open Northwest, PNSQC, and ASQ's Certified Software Quality Engineering program - becoming one of ASQ's first CSQEs. As an ACM and Agile Alliance member, his current interests include applying best practices – including appropriate agile methods – at McAfee, Inc.

What is “Cost of Quality” and Why You Should Care

What is CoQ?

Cost of Quality (CoQ) is a framework that provides answers to these questions:

- How much does poor quality cost us?
- How much are we investing in quality?

The answers to those two questions allow us to express quality in terms that executives immediately understand: profit and loss. Once engaged in that conversation, executives view quality differently – as a strategic business driver.

In short, CoQ shows how much money is spent on quality; it is sound both in theory and in practice,¹ it requires quality cost assessments,² and it moves people to action. The total cost of quality (TCoQ) equals the costs of assuring quality (**quality system costs**) plus the costs of responding to poor quality (**failure costs**).

Definition: $TCoQ = \sum (\text{quality system costs} + \text{failure costs})$

In stark contrast to software quality costs, manufacturing costs are extremely low and that is reflected in low prices of hardware. Software companies are decades behind in adopting the techniques that enable low prices. True, software development is different from hardware manufacturing³ but CoQ measurement is applicable to the software business – without modifications.

What is QPIP?

Throughout this paper you will see references to “QPIP.” It is shorthand for “quality and process improvement projects.” It reflects the strong link between good processes and high quality.⁴

Managing quality costs requires combining CoQ with a QPIP program. CoQ quantifies your improvement efforts: It is difficult to point at bugs that you prevented, but it is easy to point at a TCoQ curve that is heading in the right direction.⁵ See Figure 1.

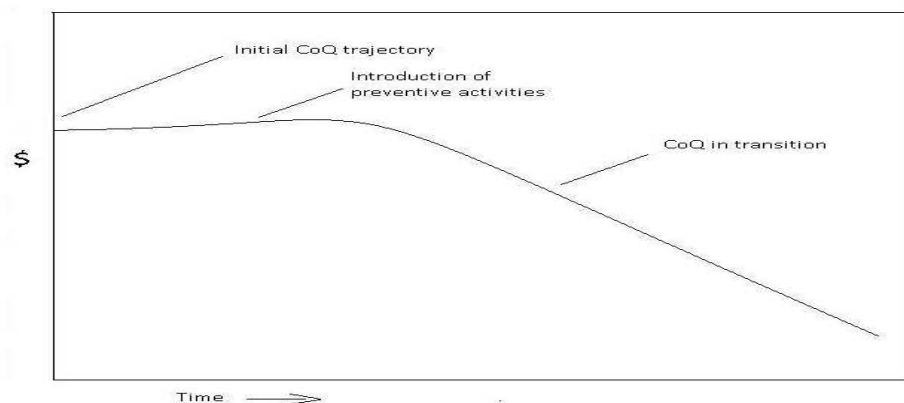


Figure 1: Over time, prevention yields lower Total CoQ

¹ See Jack Campanella's "Principles of Quality Costs", 3rd Edition, 1999, ASQ Quality Press.

² Several high-profile companies actively track quality-related time and expenses with CoQ Systems. See Part III.

³ Herb Krasner describes the salient difference elegantly in his 1998 Crosstalk article "Using the Cost of Quality Approach for Software." Software, because it is soft, can more readily be modified – it's malleable.

⁴ The more common term "software process improvement" (SPI) omits that link – and without it, executives can view process improvement as an easily axed academic exercise - not a strategic driver. To have political capital, you must be part of the solution.

⁵ If you don't track TCoQ, you will find yourself relying on anecdotal evidence to justify your improvement efforts.

Here's the first pitch for your executives:

This is the value proposition of quality:

Strategic: World-class products and services are profitable and worthwhile

Operational: Failure prevention methods are easy to use and effective

Capture this in your elevator talk.

The goal of a proactive CoQ program is to minimize failures by using industrial-strength preventive techniques.⁶ The alternative to actively managing your quality costs is reacting to failures after they occur, enduring chronic pain and constantly playing catch-up. You can either manage your quality costs by investing in prevention or hope for the best, triage incoming problems, and get further behind.

Why You Should Care

Quality Analysts, Process Engineers, and Testers: Are you doing things of little or no value? Why? Can you tell your manager? Do you always ask yourself, "How could we have caught that problem earlier?" or do you wait for others to ask that? Do you always ask, "How could we prevent that sort of thing from happening again?" Can you take actions based on your answers?

Quality Managers: How effective are you? How effective is your team? How do you know? Are you leaving "gold in the mine"? How much time of your team's time is simply wasted?

Whatever your role in software quality, you can move quality forward in your organization by actively managing your quality costs. But it is not free. Indeed, your TCoQ will *increase* before it decreases but then it can decrease dramatically – and that's the goal for you and your executives.

The software industry is leaving gold in the mine.⁷ With a few notable exceptions, we have the opportunity to greatly increase profits by adopting the lessons from other industries. In his must-read book "Quality is Free" [Crosby79] Crosby says that the TCoQ is *at least* 20% for companies that do not track CoQ. My experience is that TCoQ in software companies is roughly twice that of manufacturing companies and Krasner's excellent survey supports that belief. In [Krasner99], he concludes that 10-70% of the global software development budget is spent on quality.

Here's the next pitch to the execs: **As you address root causes by proactively managing your quality costs, you...**

- **eliminate entire families of defects,**
- **improve your products and services,**
- **increase your organization's profit and market share, and**
- **protect your job and many others.**

Minimizing mistakes in your sphere of influence helps your managers succeed. Reducing costs responsibly also helps our industry serve society better through lower prices, higher reliability, and better performance.

QPIP is also good for you professionally. The nature and impact of your work changes: It becomes productive and predictable, routine and dynamic, fun and scary. You have more interpersonal interactions but fewer interruptions. You collaborate often and feel a strong dedication to cause. You spend less amounts of time fighting fires and no time covering your tail. As you reach for the brass ring – world class quality – you spend less time hassling with bugs and more time on value-added activities, you gain experience with high-ROI projects and your influence and marketability grow. Testers may find careers evolving into process engineering, auditing, management, or quality systems analysis. Managers may

⁶ This is also the essence of lean manufacturing. See RAND article: http://www.rand.org/pubs/research_briefs/RB80/index1.html.

⁷ Dr. Joseph Juran used this phrase to sell CoQ to management in his Quality Control Handbook, McGraw Hill, 1951. Other authors refer to the unseen and underreported costs as the "hidden factory".

find higher visibility and influence in the organization and quicker promotions. All will find work more enjoyable.

There will always be some need for manual functional testers but as the software quality profession is following the same trajectory as other industries (notably manufacturing, community policing, and health care), fewer testers will be needed. Quality professionals find our roles and responsibilities change as our quality organizations mature and their missions evolve (as proposed in Table 1).⁸

Growth Phase	Level	Primary Activity/Mission
initial	0	Help make the software work
	1	Break the software (find bugs)
awakening	2	Assure feature functionality / Assess functionality
	3	Assure software attributes / Assess quality
maturity	4	Improve processes / Prevent problems
	5	Audit processes

Table 1: Software Quality Organization Maturity Levels.

As you become more immersed in QPIP, your organization matures.

How to Manage Quality Costs

This section addresses the question, “Okay, how am I going to do it?” The quest is to find time. The quest starts with a charter for your active CoQ and QPIP efforts. Here’s a sample, generic QPIP charter:

Maximize ROI on available quality resources by using preventive measures to eliminate entire families of defects and waste.

Next, familiarize yourself with Krasner’s research into ROI on process improvement efforts.⁹ Any effort that returns a 20-30% premium is worthwhile but efforts that return 200% or 1000% are slam dunks. You will have executive attention.

Convincing management to undertake serious quality improvement requires learning some jargon. The next section introduces the main terms you’ll need.

Definitions: System and Failure Costs

Many software activities are directly related to quality. Active CoQ addresses those quality-related activities (see the Appendix for examples). Quality-related costs and activities are normally divided into two broad categories: *System* and *Failure* which both have two subcategories as shown below and in the Appendix.

System costs ~ Those things you choose to do to prevent defects or detect them.

Prevention ~ everything associated with avoiding failures and waste.

Appraisal ~ everything associated with test and measurement.

Failure costs ~ Those things you must do to recover from creating or shipping defects:

Internal failures ~ defects found *internally* (before shipping)

External failures ~ defects found *externally* (by customers or others after shipping)

With those basic definitions, we can move on to our improvement strategy.

⁸ While much is written about organizational maturity and its link to software quality, the maturity of quality organizations is largely absent from the literature other than the current paper and <http://www.aof.mod.uk/aofcontent/tactical/quality/downloads/gmmm.doc>.

⁹ See Krasner’s 2001 report <http://www.compaid.com/cailInternet/casestudies/krasner-spiROI.pdf> for ROI data in the public domain.

General CoQ Improvement Strategy

The necessary and sufficient tasks to manage quality costs, then, are these:

1. Get support – find a champion
2. Assess your major quality activities and other quality costs
3. Propose/Select improvement projects
4. Execute and monitor
5. Repeat and institutionalize

Note: Size matters. Small organizations can just do it – review quality successes and failures at each planning meeting – daily, sprint/scrum/iteration, and release – to take corrective actions. Larger organizations with committed resources can also transform into a world-class¹⁰ software shops by managing quality costs but the journey is necessarily more structured and formal. The rest of this section focuses on those larger shops.

Get Support – Find a Champion

First, you need support for making quality costs visible and for implementing corrective actions. So find a senior executive responsible for allocating money to support the effort. She needs to represent your efforts to other executives and defend your efforts over time. You'll need a great elevator talk and a perhaps a presentation about eliminating families of defects and increasing market share through better quality. It may be a tough sell: when quality costs are high, research funds are slim.

If you can break through her expectation of QA to just to find bugs, she may invest in defect prevention. But a certain organizational maturity is required to embark on a quality improvement project even when cost avoidance is the goal. Gaining a champion can be tough but once you clear this hurdle, you are well on your way to continuous improvement, higher profitability, and better days.

The first thing you need to sell to management is that you feel strongly that you can help her save tons of money. All you want initially is to conduct a CoQ survey to show her how much your organization is spending on quality. You can tell her later that you want to do those surveys repeatedly.

Assess Your Major Quality Costs

There are two main ways of determining your quality costs:

- 1) A series of surveys or
- 2) A more formal tracking system.¹¹

Warning: Your tracking system cannot become part of the problem. It is not a formal accounting system. It needs to be lightweight and show the quality costs in each category. Because quality improvements result in large improvements, your CoQ system can paint broad strokes.

Here are two low-cost options for the initial CoQ survey:

- Zoomerang
 - Pros: quick and cheap
 - Cons: spotty results, relies on recollections, responses may not be statistically significant
- Interviews and Excel
 - Pros: more detailed, better consistency
 - Cons: labor intensive so more expensive

¹⁰ Reviewers have suggested that “world-class” is ambiguous. For this paper, it equates to Crosby’s “Quality Certainty” stage and the equivalent. For background information, see:

- [Crosby’s Stage V - Certainty](http://en.wikipedia.org/wiki/Quality_Management_Maturity_Grid): http://en.wikipedia.org/wiki/Quality_Management_Maturity_Grid
- [Bhote’s Ultimate Six Sigma](http://www.amacombooks.org/book.cfm?isbn=9780814407592): <http://www.amacombooks.org/book.cfm?isbn=9780814407592>
- [Zero Defects](http://www.managers-net.com/zerodefects.html): <http://www.managers-net.com/zerodefects.html>
- [TQM](http://en.wikipedia.org/wiki/Total_Quality_Management): http://en.wikipedia.org/wiki/Total_Quality_Management
- [Project Management for Adults](http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=589226): http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=589226

¹¹ Such as an Activity-Based Costing (http://en.wikipedia.org/wiki/Activity-based_costing)

Cost management systems are more rigorous and statistically defensible but they are more expensive. Put such a system on your wish-list for later – after your QPIP efforts have shown their worth.

Your first CoQ survey could use Zoomerang and ask just four questions of randomly selected employees:

1. How many hours did you spend last week on field failures and customer complaints?
2. How many hours did you spend last week diagnosing and fixing bugs?
3. How many hours did you spend last week doing testing and evaluations of any kind?
4. How many hours did you spend last week preventing problems from reoccurring?

Warning: You may get questions about how you intend to use the data. Assure the surveyed people that the survey results will be free from all identifying information. You can also expect widely divergent definitions of “field failures” and other terms. Include some examples in your survey.

Your results may vary but do not be surprised if the results look something like those in Figure 2.

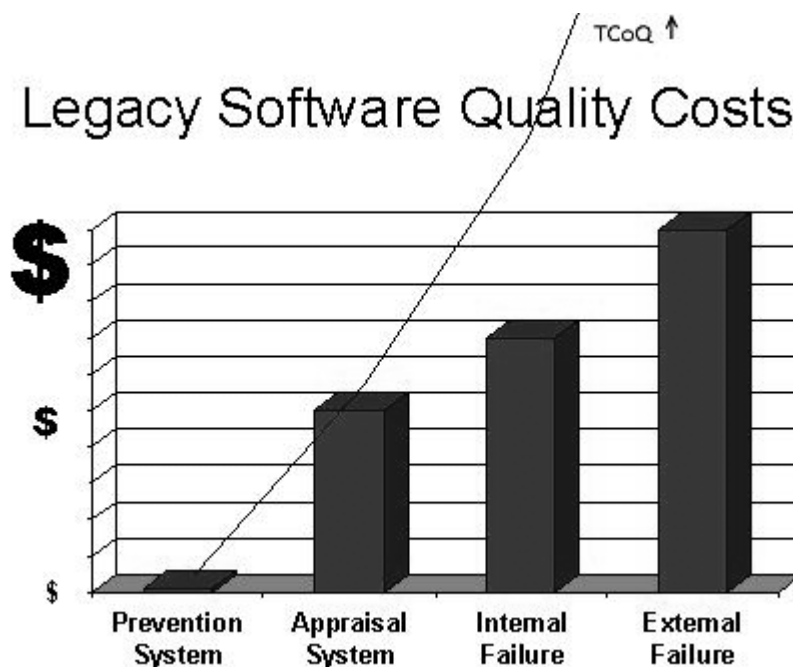


Figure 2: Typical legacy software quality costs – note that TCoQ is off the chart

Propose/Select Improvement Projects

The next thing is to select an improvement target. Your people usually know what changes can lead to big improvements. Your bug tracking database can be extremely useful in finding opportunities and patterns of failure. Running a few queries uncovers low-hanging fruit. Since bugs tend to cluster, you may identify components that need refactoring.

Warning: At the beginning, there will be lots of ideas and consensus may be hard to reach. Stick with it. Like any other team a QPIP team must form, storm, and norm before it can really perform.¹² Focus on those things that are causing the most pain for your customers and other stakeholders. Do not try to boil the ocean. You cannot address every concern at once nor find a solution that will fix all problems. Start

¹² Attributed to Bruce Tuckman, 1965: <http://en.wikipedia.org/wiki/Forming-storming-norming-performing>

with important issues and keep it simple. You need to show early successes, collect organizational capital, not cure everything in one go. QPIP requires continuously honing ones soft skills.

You are looking for your biggest problems for which you can take corrective actions and generalize the solutions. Let's take those in order...

Identify your biggest problems

Your champion can be an excellent source for improvement ideas. Involving her has the additional benefits of aligning your activities with company goals, keeping the discourse in management terms (cost avoidance), and keeping senior management in the loop.

Sometimes the target is painfully clear, for instance when your company has suffered an embarrassment in the market. But more often you will have multiple improvement options. In those cases, this is a useful QIP planning method:

1. Brainstorm the important problems – everyone contributes at least one “if only X, we could have Y”,
2. Affinity grouping – wherein like items are grouped together, and
3. Perform a Pareto¹³ analysis – wherein you converge on the “significant few” problems.

Two warnings when discussing patterns of failure:

1. Studiously avoid talking about people. As Deming taught us, 85% of root causes are process-related and only 15% are people-related. Direct the discussion toward process improvement even if everyone knows that Joe needs training.
2. Use the term “failure” infrequently. In a supportive, non-threatening way have the team suggest *improvement ideas*. That's much less threatening than talking about patterns of failure. Make it clear to everyone that you are not looking for scapegoats. Instead you are looking to refine (or implement) effective processes.

Take corrective actions

Once you have selected the one or two most important problems, dig for root causes. Ask this Deming-inspired question “What part of the process or system allowed this problem to happen?” Answering that question allows you to make process improvements that eliminate entire families of errors.

The term “corrective action” means effectively remediating root causes. Too often people feel that they have taken corrective action when they have found a design or coding fault. But without addressing the causal factors, other instances of that same fault will occur – and no long-term benefits are gained.

Figure 3 shows one useful framework for brainstorming possible root causes.

¹³ Also known as the 80/20 rule. See http://en.wikipedia.org/wiki/Pareto_principle.

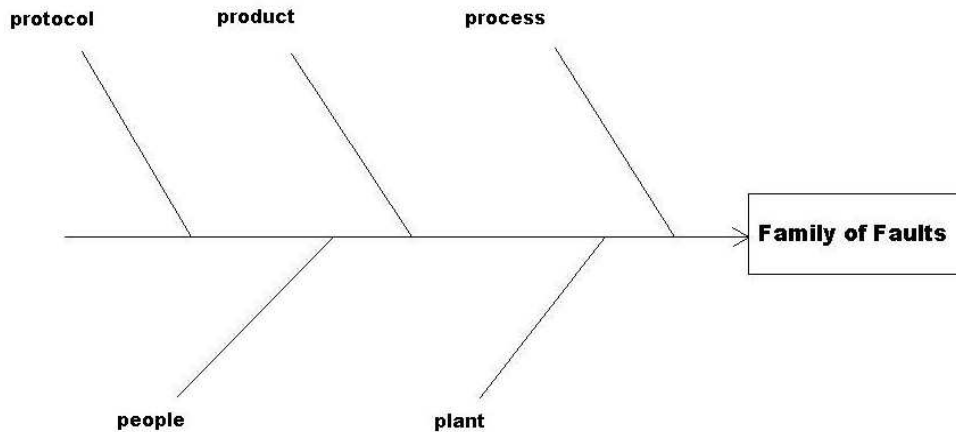


Figure 3: A generic cause-and-effect diagram (aka Ishikawa or fishbone diagram).

Be sensitive to the differences between symptoms, problems, causes, and solutions. In IEEE terms:

- Symptoms are FAILURES. These are the effects manifested during product use.
- Problems are FAULTS. These are the underlying issues with the code or the design.

Causes are those things that allow the faults to occur. These root causes are the real targets of CoQ/QPIP. Identifying the most influential root causes is vital. Labeling each cause as *necessary* or *sufficient* allows you to rank which causes to address first. Careful selection of the causal factors leads to the best ROI.

Note: You cannot address *every* causal factor. For instance, some causal factors may be outside of your sphere of influence others may require more resources than readily available. Using the S.M.A.R.T. goal-setting technique will help. To increase your effectiveness your atomic CoQ/QPIP goals, like all good goals, should be **s**pecific, **m**easureable, **a**ttainable, **r**elevant, and **t**ime-boxed.

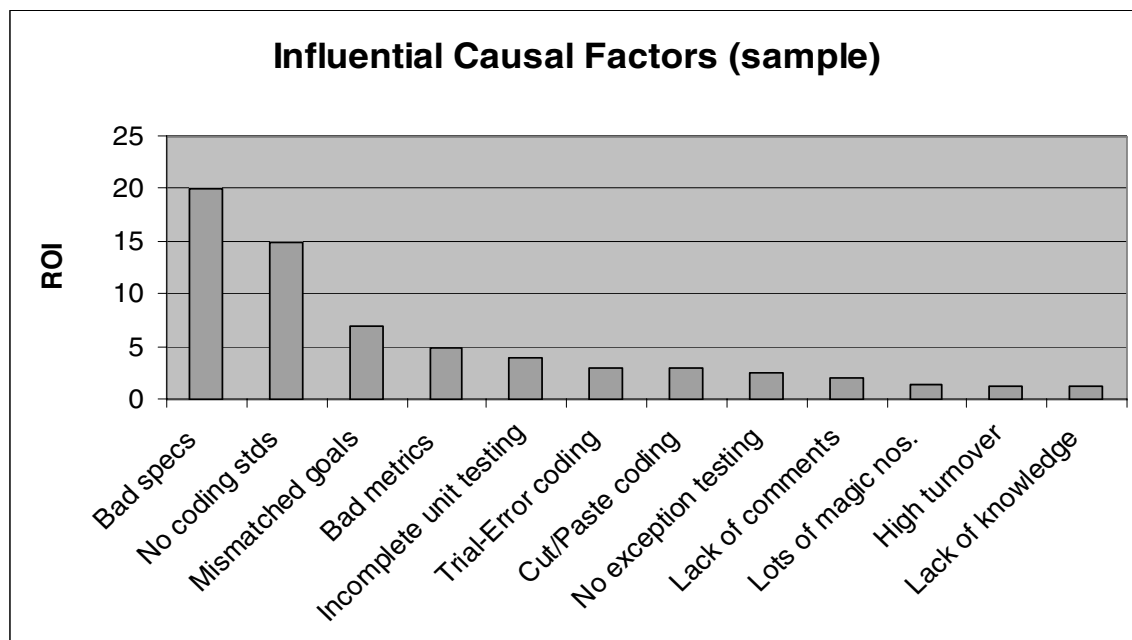


Figure 4: A sample Pareto chart of causal factors

Generalize solutions

Okay, you have identified and ranked your problems, determined the most influential causes, and created a corrective action implementation plan. But you're not done. Just as *faults* can cause multiple *failures*, *solutions* can solve multiple *root causes*.

Before you present your plan to your management champion, consider whether your proposed corrective actions will have multiple benefits. That is, will you get even better ROI than documented in your plan? If so, you have two options: 1) a simple notation in your plan that other beneficial side effects are probable or 2) revisit and expand your plan. This latter option takes you back to the affinity grouping step (above).

Generalizing the solutions helps you and your sponsor keep the overarching organizational goals in focus and further fund your CoQ/QPIP efforts.

Incidentally, the time you spend discussing CoQ/QPIP with your champion, the time you spend constructing and administering the surveys, the time you spend ranking problems and solutions, and the time you spend reading this paper – those are all Prevention costs. With respect to the late Phil Crosby, quality is not free: Managing quality costs is an expense but the payback can be huge.¹⁴

Execute and Monitor

Nothing will kill a QPIP quicker than lack of follow-through. Collaborate with your champion to assure that the action plans you design are well-received in your organization. Train your people how to respond to questions about the CoQ/QPIP. Keep all stakeholders fully apprised and involve as many of them as possible. Definitely involve the affected people when defining and implementing a new process.

A monitoring mechanism must be part of your action plan. Once your changes are implemented, stay in close touch with the affected people. Be prepared to change your solution. Be prepared to start over.

For you as a quality professional, once you've implemented a few process improvements, 1) you will be an effective change agent with corresponding growth in influence and 2) your negotiation and facilitation skills will be improving.

Repeat and Institutionalize

A successful CoQ/QPIP program is a journey that never ends. But one major milestone is met when QPIP is institutionalized... when it becomes part of the organization's DNA.

One way for large organizations to close the loop – or “ratchet” as Juran called it – is to establish a quality steering team. Your quality champion can lead that team but it also must include your most-senior executive. Their mission is to integrate quality/process improvement into every employee's work. They fund and focus QPIP teams and give the CoQ/QPIP program legitimacy. The very existence of a top-level quality steering team tells the world “we are serious about quality.”

One successful model is that the steering team meets regularly with smaller action teams (2-4 members), empowers them and holds them responsible for continuous improvement in certain domains such as the Capability Maturity Model's Key Process Areas.

Implementing Quality Circles is another way to institutionalize quality improvement. This, though, is a major undertaking that can only succeed when the entire organization appreciates the value of QPIP. It is not a year one objective in large software organizations.

¹⁴ The claim that “quality is free” arises from Phil Crosby's recognition that defect prevention pays for itself many times over.

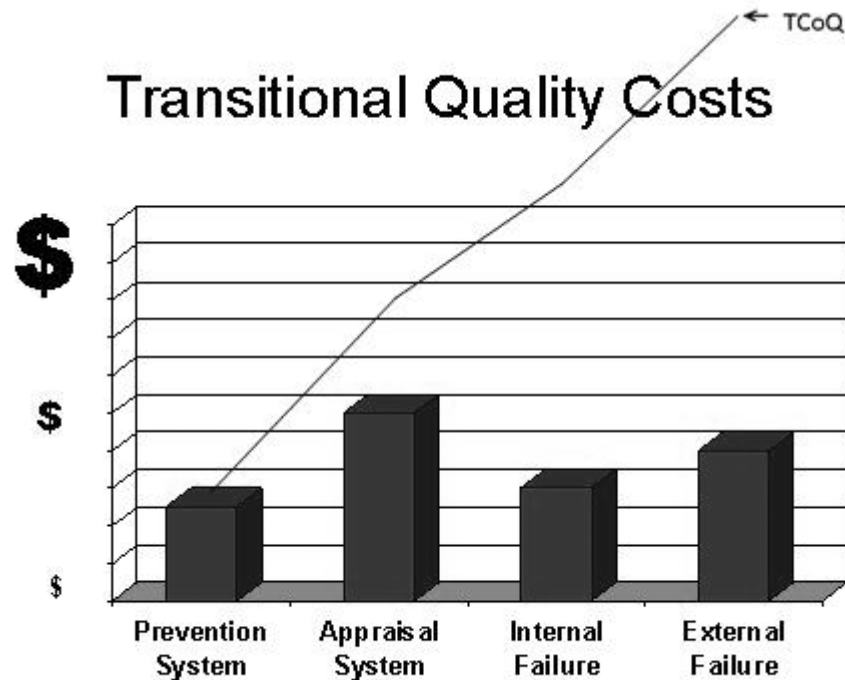


Figure 5: Possible quality cost profile after one year – note TCoQ

Let us fast forward one year beyond your initial TCoQ survey: Your executives are seeing that QPIP is profitable. The quality cost profile has changed considerably (Figure 5), people are happier, schedules are more predictable, and estimates are better. Products are selling better and profit margins are improving. The testing department is no longer frantic and they are now working 40 hour weeks. TCoQ reports are now reviewed at every board meeting. The company is awakening to the strategic value of quality.

You might think that finding good improvement projects would get harder after one year. But it doesn't for two reasons:

- 1) The environment keeps changing - processes don't fit forever, the competition doesn't stand still.
- 2) Your culture is changing. With quality as a strategic driver, people are seeing real improvements in their operations, they are re-invigorated and engaged. This generates new improvement ideas.

Learn, Practice and Preach

Beat the drum. As you get feedback about your process improvements, share that with the stakeholders. Let them know how much money you have saved the organization. That builds your political capital and makes further improvements easy to justify. At this stage of the software industry's evolution, the consummate quality professional is an educator and a practitioner.

Your organization's new focus on preventive actions allows you to accelerate the improvements. What does it really mean to invest in prevention? It means:

- Training: read books, go to conferences, lead discussion groups, mentor one another
- Product planning: involve customers in product evolution including end-of-life planning
- Quality planning: expand testing to include all quality attributes, establish tradeoff protocols
- Documenting: move away from tribal knowledge and its reliance on warriors and heroic efforts
- Analyzing failures: use each failure as a learning experience to improve processes

Push the Envelope

Keep running... Even the Toyota Way is constantly changing. Read, do experiments, collaborate, develop and apply new ideas. Success breeds success. Talk to your internal and external customers and suppliers. Know their needs and methods. Think in terms of systems and processes. Talk to your sponsor about sharing the wealth. At least one company shares the cost savings accrued during the first year with the responsible contributors.¹⁵ Monetary incentives aid the institutionalization and keep the focus on money – the language of management.

After you spend a couple years seriously removing root causes of failures, you don't have as many product failures. The code is cleaner. In the Bach brothers' parlance – you will have more T and less BS (more *testing* and less hassling with *bugs* and *setup*)¹⁶. Do you need to test your software as much? This is a non-trivial question. Would reductions in testing introduce too much risk?

The manufacturing world moved to small sample sizes long ago. A modern high tech manufacturing cost profile looks like Figure 6. Krasner¹⁷ tells me that General Dynamics Defense Systems, Motorola Cellular Systems, Qualcomm, Honeywell Building Control Systems, and Xerox Office Products have thriving CoQ programs. How long will it take us software people to catch up with the hardware people?

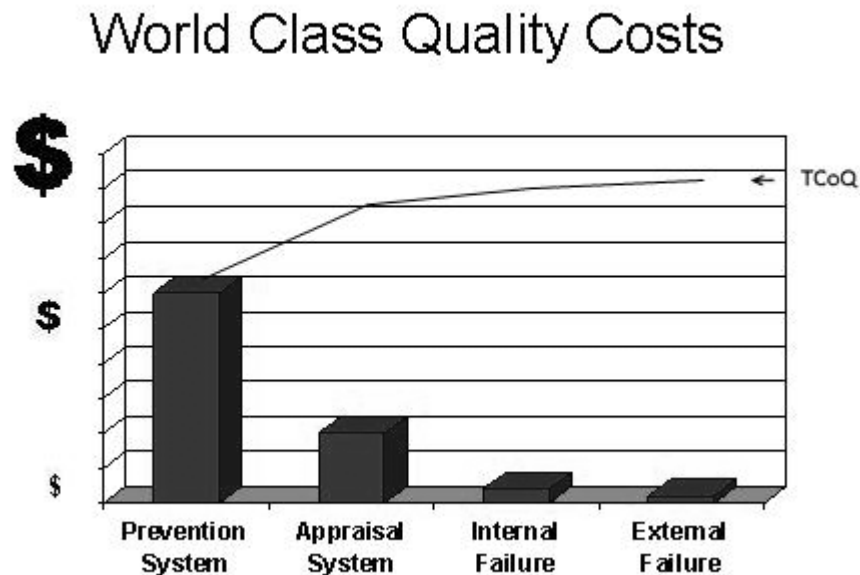


Figure 6: Quality system costs far exceeding failure costs – note TCoQ

¹⁵ This is another cost of quality improvement – another Prevention cost – one that the company is happy about.

¹⁶ Presented by Jonathan Bach at a Software Association of Oregon event, Wilsonville, 2007.

¹⁷ Private communication June 2009. See also "Accumulating the Body of Evidence for the Payoff of Software Process Improvement": <http://www.compaid.com/cailInternet/casestudies/krasner-spiROI.pdf>

Conclusion

You will find time for quality/process improvements once you have started the QPIP journey.

Many quality professionals are one executive decision away from tough times. But layoffs are avoidable for two reasons:

- 1) Companies that prevent problems prosper.
- 2) Companies that allow problems to reoccur are fertile grounds for QPIP.

In most software organizations, radical improvements are possible and you can contribute in ways large and small. But the improvements must be visible and CoQ tracking is an excellent mechanism for that.

Our times are marked by economic upheaval. That turmoil is not entirely outside the scope of this paper. Some current issues are definitely quality related – Ford, Chrysler and GM are playing quality catch-up to Japan's car companies. It's extremely expensive for them.

Like high tech quality, **software quality is not a matter of luck**: When you don't prevent recurrences, you are bound to see the same mistakes. That is a waste of time and talent. And if your organization wastes too much time dealing with avoidable problems, eventually it is forced into cost-cutting measures (e.g. layoffs). "Cost cutting" should have a different connotation – profitability improvement through defect prevention.

In my group, that means further reducing our incoming defect density by using more agile development methods. We have sufficient autonomy – and have made sufficient progress – to take some risks. In short, we are in the process of moving beyond being a test organization and becoming a quality organization. Just like PNSQC is a quality conference – not just a testing conference.

You Can Choose Your Future

We will always need testers

...but not as many

We always need improvement.

This isn't some esoteric or academic theory. You have practical choices to make. You can proactively manage your quality costs on a micro level. And you can influence quality costs at a macro level. When you find a good bug or when you let a bug escape to the customers, you have a chance to significantly improve your products and processes. You can collaborate with your colleagues to determine root causes and act to address them. Or you can duck; keep your head down, and hope that nobody traces the problem back to you.

If you choose the low road, you will find that you and your companies never really prosper.

If you choose the high road, you embark on a challenging, rewarding and exhilarating journey into group dynamics, organization maturation, and sustained improvements. Prevention as a way of life... When that becomes your mantra, you, your colleagues, and your company will win big.

Toyota has chosen the high road. They *want* people to find problems. They literally applaud people who stop the production line. They train people to find root causes. They continually evolve and they now sell more cars than anyone else. It is time for the software industry to catch up.

Acknowledgements

The author would like to thank these people for their thoughtful reviews and/or other feedback: Cynthia Gens and Jonathan Morris – my official PNSQC Reviewers; Jim Fudge, Patt Thomasson, Josh Eisenberg, Ellen Ghiselli, Patricia Lee, Justin Kelly, and Robert Dufur – McAfee colleagues and friends. Thanks to all for your valuable feedback.

Appendix – Some Quality-related Software Activities

<i>Quality Systems</i>		<i>Failures</i>	
Prevention System	Appraisal System	Internal Failure	External Failure
training reading seminars conferences formal education failure analysis corrective actions quality improvement process improvement quality planning product planning documenting commenting code writing process docs joint specs definition QPIP capacity planning	test planning acceptance testing verifications validations test planning inspections design reviews code reviews assessments audits unit testing	technical debt code corrections design corrections unexpected refactoring waste unproductive testing abandoned efforts ignored plans debugging added operations rework: re-planning re-designing re-verifying re-validating re-inspecting lost time unneeded testing waiting for meetings waiting for hardware cancelled projects	technical debt field failure phone support triage defect management patches and hot fixes lost market share firefighting damage control lost business bad press litigation / settlements returned products

The four quality cost categories – Prevention, Appraisal, Internal Failure, and External Failure – are sufficiently distinct and you can determine the quality category to which an activity belongs by asking: “What is the **primary reason** for doing this?”

For instance, the reason for a refactoring task may be to eliminate structural weaknesses and, if so, it is a prevention cost.

If the same refactoring is done to correct several reported bugs, it is a failure cost. If the bugs were reported by customers, it is external failure; if reported by a testing group, internal failure.

If the same refactoring is done to abstract useful classes, it is a normal development cost and not something to include in the TCoQ.

If you are extending your test harness by refactoring locally-written library routines, it is an appraisal cost because the primary reason for doing it is to test the customer-deliverable software.

But don't get too caught up with categorizing your quality costs. Remember that CoQ tracking is a broad brush approach – it is not an accounting system that needs to zero-out to the penny.

Also, many activities have multiple quality-related reasons but, again, don't get bogged down in arguing minutia. Doing so takes energy from the goal of reducing overall quality costs.

Bibliography

Crosby, Phil; 1979, Quality is Free, Penguin Books

Juran, Joseph; 2005, Juran's Quality Handbook – Fifth Edition, McGraw-Hill

Kaner, Cem; 1996, "Quality Costs Analysis: Benefits and Risks" <http://www.kaner.com/qualcost.htm>

Krasner, Herb: 1999, "Using the Cost of Quality Approach for Software",
http://sunset.usc.edu/cse/pub/event/archives/pdfs/Herb_pt2.pdf

Krasner, Herb: 2001, "Accumulating the Body of Evidence for the Payoff of Software Process Improvement" <http://www.compaid.com/caiInternet/casestudies/krasner-spiROI.pdf>

The Search for Software Robustness

Dawn Haynes
dhaynes@perftestplus.com

Abstract

International Software Testing Qualifications Board (ISTQB 2007) defines robustness as “the degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions.” While this definition is straightforward and reasonable, this separate classification seems to indicate that during normal operation software would never encounter unacceptable or unexpected inputs, experience uncooperative interfaces, or be impacted by limited resources. However, these conditions are exceedingly common and often pose problematic situations for software users.

Robustness testing is an important and often overlooked area of testing, especially when test teams are over-tasked and under-resourced. Testing to verify that requirements are met is necessary, usually commands the highest priority, and often leads teams to focus on trying to prove that the software works instead of trying to demonstrate the ways in which it doesn't work. Positive or “happy-path” testing tends to cover a small amount of the system's code base, leaving many code pathways virtually untested. Targeting tests toward how the software handles unexpected events, errors and failures can go a long way to shrinking the amount of untested code, and reducing the risk of exposing users to fragile software.

In order to truly improve your software's robustness, you need the mission, the strategy, and the support in place to be successful. Without support, a vigorous robustness testing effort could just churn out a bunch of bugs in the defect database that don't get fixed (which would be a big bummer, and probably a waste of time!). This paper explores several techniques to aid you in developing or enhancing your strategy for evaluating the robustness of your software.

Biography

Dawn Haynes is a senior trainer and software testing evangelist currently focused on improving the maturity of testing within software development teams by increasing the awareness of testing concepts, processes, techniques and tools, through education and assessments.

Dawn has over 25 years of experience with systems ranging from corporate infrastructure to desktop productivity tools in the domains of insurance, healthcare, and commercial software manufacturers of font technologies, OCR products and test tools. Dawn has held positions in software development, technical writing, systems administration and IT support, customer service, field engineering, quality assurance, course development and training.

Dawn is also a Director of the Association for Software Testing (www.associationforsoftwaretesting.org), a non-profit organization dedicated to advancing the understanding and practice of software testing throughout the industry. Dawn holds a B.S.B.A. in MIS with a concentration in computer programming from Northeastern University in Boston.

Copyright © 2009 PerfTestPlus, Inc. All rights reserved

Published at the Pacific Northwest Software Quality Conference 2009

Introduction

A common “standards-based” definition for robustness in software is the degree to which a system operates correctly in the presence of exceptional inputs or stressful environmental conditions, but what exactly is an “exceptional” input or “stressful” condition? I frequently find software that fails to operate correctly even when I believe my inputs are valid and the only stressful condition that exists is me fighting off the urge to pound my computer with a sledgehammer. So what’s the big deal about robustness if it’s beyond the realm of the valid inputs, expected usage and well-behaved operating conditions?

In the very real world of software use, unpredictable usage and chaotic environments can cause even well-designed and reasonably tested systems to become unstable, corrupt data or even crash without warning. Designers and developers often struggle to balance the desire to give users flexibility against the need to prevent users from causing trouble for themselves or the system. From a testing perspective, testers try to mimic or explore real usage scenarios to verify what works, but often don’t push the envelope to find out what doesn’t work due to scope or schedule constraints. Software that is weak, fragile and falls over at every turn will not only impact users, but it can stall or even block the best testing efforts from making progress.

This is where robustness testing can add value, but getting started can be a challenge. I’ve outlined what I believe are some key elements that need to be evaluated and provided suggestions to enable you to integrate robustness testing into your projects quickly and successfully.

1. Why does robustness matter?

Software gets a bad rap sometimes, and sometimes it’s even well deserved. Take Microsoft, for example. Who doesn’t enjoy sharing their favorite poke at the software giant? How about those Mac vs. PC ads on TV, right? Says it all, doesn’t it? One would think that a tarnished reputation, unenviable perceptions, aggressive competition, or potential public embarrassment would motivate software providers to be keen on avoiding robustness flaws. Within projects however, there are often competing priorities and goals that eclipse some of the fundamentals, such as “software is for users”. It might be worth getting to “know thy user, for he is not thee.” (Platt 2007)

I truly believe that when software is so frail and fragile that there is only one small set of safe scenarios to execute, the software’s value is exceedingly compromised. Users know what they want to accomplish, but don’t always know the perfect set of inputs, the preferred navigational pathway or sequence of operations, so it’s easy for them to cause problems with software. When users don’t use software exactly as imagined by analysts, architects, designers and programmers, they shouldn’t necessarily be punished by having their data taken away, or their work trashed.

Most users appreciate gentle guidance, informative and useful error messages, and sometimes even prefer a friendly warning of impending doom over the element of surprise. The newer Microsoft Office applications have such a message now, which I much prefer over the blue screen of death. The kindly message says something like “I’m in trouble and things are only going to get worse. If you like your work, now would be a good time to save your data and get out of Dodge.”

Truly fragile software can impact a project’s success as much as poor performance. In both cases, all the features and functions could be perfectly implemented, all the requirements met, all the contractual obligations fulfilled, and users can still end up wildly dissatisfied. Robustness testing strives to answer the question, “How likely is it that real users, under real conditions, with all of the seemingly arbitrary, random,

and unexpected things that happen while they are using the software, will have faith in the system as opposed to complaining about how it is unusable, broken, unreliable, or just plain bad?” If avoiding this kind of “bad” seems like a good idea, you have arrived at the beginning of your search for software robustness.

2. Why is robustness often overlooked?

Avoiding robustness testing is easy! Just don’t require any evaluation of it. There is a saying in management training courses: doers do what checkers check. Not surprisingly, the inverse is at least equally valid; doers don’t do what checkers don’t check. This is the simple reason most teams don’t do robustness testing – no one tells them to and no one checks to see if they have.

The omission of robustness testing can also be an artifact of using certain testing methods and approaches. Requirements-based testing, for example, is a very popular testing approach and depending on how “requirements” are developed, it can be very thorough and effective. However, one possible downside of this approach is a tendency to exclusively focus on specific features, functions, user-interface elements and other implementation details. When this occurs it becomes extremely easy to miss the opportunity to define quality factors (Guckenheimer and Perez 2006), such as: reliability, general performance and responsiveness, intuitiveness, consistency, and robustness.

Another easy way to avoid hitting a lot of robustness problems is to rely heavily on many short burst tests (manual or automated) that focus on evaluating one thing at a time without considering how users interact with the system over a period of time. Mimicking real user patterns, durations, and lapses can easily expose robustness issues. Here’s an example from my own experience of one that got away:

We shipped a new version of our API toolkit that allowed software vendors to incorporate our technology into their products. One vendor embedded our solution as a core function of their flagship commercial product that performed automatic fax routing which was implemented as a Windows server-based solution intended to run 24/7. We never tested our software for long durations because we didn’t use it that way when we built our own commercial products, so we were quite surprised by the vendor’s report that our component’s response time that initially was only a few seconds, later degraded to 30 minutes or MORE. Investigation indicated that the problem only occurred when their software was running for more than 24 hours. Root cause analysis revealed that the problem was due to an unconfigurable operating system setting that automatically (and without remorse) demoted the priority of services and processes running for more than 24 hours. Based on our architecture, the only workaround our customer could give to their customers was to restart their “set it and forget it” solution every 24 hours. Fun times!

It’s easy to overlook what you don’t know, can’t anticipate or don’t understand. Awareness of robustness issues is a key factor in successfully trapping them as early in the software development life cycle as possible. One way to put robustness on the map as a detectable concern is to educate the team and start publicizing real and or potentially costly instances as they are detected by testing or encountered by users.

3. Educate the team

Robustness issues can block testing progress and prevent users from getting value from software. It is crucial to help the team understand these issues in terms of real impacts: project costs, delays, support costs, lost sales, bad press, etc. Trying to convince project level stakeholders that the sky is falling without hard data is going to be difficult. Sometimes it’s possible to use relevant industry stories of real pain to make the point without having to experience problems first hand. I recommend scanning technical

news reports and journals, blogs and discussion groups for robustness related fiascos to share at a project kickoff meeting. (But don't forget to bring a spoonful of donuts to help the medicine go down!)

Here are a couple of real-world examples that illustrate types of problems your team might want to avoid. The first example could be a reliability issue, or could be related to robustness. From the very helpful error message below (see Fig. 1), it's hard to tell. It occurs frequently when I attempt to send an email from Yahoo! Mail, but even *more* frequently (about 80% of the time) when there is an attachment in the message. When I speak with technical support, they are unwilling to disclose specifics about the problem, but my frequent calls and unrelenting queries have revealed that error #14 is a server-side problem that only they can fix. From my end I am unable to determine if there is something invalid about my emails, attachment contents or format, or something awry in my local configuration. I am of course evaluating replacement email solutions as I write this.

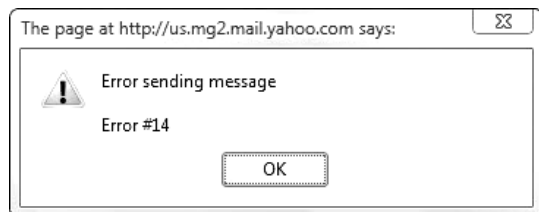


Figure 1

The second example is more clearly a robustness issue. I am confident of that because my laptop was exhibiting exquisite stress when forced to run for a long period of time (several days without a reboot, and many hibernations in between). The machine would frequently freeze, crash every running application, and then the desktop would seize up so badly that Task Manager wouldn't run. During one of its fits, the operating system spit out one of the most hilarious error messages I have ever seen (Fig. 2). In between hysterical laughter and plans to commit hari-kari, I frantically tried to pull a screen shot and was successful! I was amazed it worked because the next error I got was "unable to run MSPAINT.EXE – not enough quota." My choice is simple – forgo the convenience feature of hibernate, or investigate other machines and operating systems that aren't so fragile they forget their own children's names!



Figure 2

Another way to facilitate quick progress is to give people a new vocabulary to help identify, discuss and classify issues. Publishing a glossary or posting a set of definitions is a good start. Here a few excerpts from the ISTQB Glossary which contains a collection of terms from a wide variety of industry sources including books, articles, jargon in common use as well as many published by international standards boards (ISTQB 2007):

robustness: The degree to which a component or system can function correctly in the presence of invalid inputs or stressful environmental conditions. [IEEE 610]

reliability: The ability of the software product to perform its required functions under stated conditions for a specified period of time, or for a specified number of operations. [ISO 9126]

fault tolerance: The capability of the software product to maintain a specified level of performance in cases of software faults (defects) or of infringement of its specified interface. [ISO 9126]

Mapping examples to the definitions is also a way to help people more quickly correlate and connect the terms with actual issues. It might even be useful to add classifications to your defect database so problems can be tagged for proper triage, root cause analysis and potentially to feed future process improvements.

Think of creative ways to internally champion the idea of focusing on robustness. Imagine running an internal campaign for “Captain Robustness” by spamming inboxes and pasting hallways and common areas with banners that read, “Robustness is Job #1!” Or maybe it’s job 2 or 3, but you get the idea.

Put together a library of resources or a recommended reading list of articles and books that help refocus people on the team and address key knowledge and experience gaps. I enjoy sharing books that help people to think from a user’s perspective more easily, and those which help people understand the myriad challenges in software testing better. On my current short list of favorites, I highly recommend:

- “Why Software SUCKS...and what you can do about it” by David S. Platt
- “Bad Software: What to Do When Software Fails” by Cem Kaner, David Pels, and David L. Pels
- “Perfect Software and other illusions about testing” by Gerald M. Weinberg
- “How to Break Software: A Practical Guide to Testing” by James A. Whittaker

Build your own handbook or guide for robustness topics. I found an excellent example published as a Technical Note by Carnegie Mellon University (Cohen, Plakosh and Keeler 2005) named “Robustness Testing of Software-Intensive Systems: Explanation and Guide.”

It is often difficult to make progress without high-level directives and clear management support for evaluating and improving robustness. One indicator of a stalled effort is when robustness-related bugs discovered in testing are routinely deferred or simply rejected as “not a defect”.. Be on the lookout for some key phrases such as, “it functions as designed” or “the system operates according to specifications.” Another favorite is “a user would never do that.” In my experience, I’ve learned to never try to predict what a user will or will not do with software. I instead work to evaluate just how much trouble users can get themselves into, and help the project team identify how much “we” care about it.

4. Develop a test strategy that’s easy to implement and adds real value

Some decisions are going to need to be made about where to strike with robustness tests. There are so many things to try I get all tingly thinking about it! I’m a natural stress and break tester. Geez, I almost never want to know if something works. And with a world of possible tests to choose from, it’s going to be useful to provide the team with some direction about scope and span. For example, developing a mission or charter to help teams understand what things are most important to focus on first.

Select high-risk scenarios or features:

One useful frame I’ve borrowed from Whittaker (Whittaker 2002) for developing a strategy to target scenarios is to think of the search for robustness like this:

1. *Functional code* accomplishes the mission of the software by implementing user requirements.
2. *Error-handling code* keeps the functional code from failing.

In other words, we like features, and we put safeguards in code to help features work without failing. So focusing tests on checking that the safeguards work sufficiently based on the priorities of your features is a great way to start.

For more ideas on organizing risk-based testing strategies, refer to James Bach's article (Bach 1999) a heuristic approach to risk-based testing. Implementation suggestions are well outlines with many checklists to reference.

Avoidance strategies:

Think about the evil-bad things that would be awesome to avoid. You could contemplate some worst-case scenarios, or as recommended by my friend Elisabeth Hendrickson, try the headline game instead. Think about the things you would like to avoid seeing on the FRONT page of the newspaper ABOVE the fold, just after you ship. Then try to make those things happen!

A good strategy should convey the direction and focus of the effort as a start. To be more complete, include tangible and reasonable goals, define a way to measure the effort and effectiveness of what is implemented, and provide a list of alternate ideas to tap should the first choices become obsolete or uninteresting.

Earlier is better:

Try kicking off a new development project with a focus on robustness as a theme. Weave the theme into documents, processes like reviews, status meetings, perks or incentives, and even geek gear! (I'm thinking of a project t-shirt that could end up being less obviously geeky than most.) With a focus on robustness from the start, it might even be possible to build robustness in instead of find out it is not good enough later.

5. Robustness testing ideas

Selecting specific tests that poke at robustness and reliability can be tricky at first. Whittaker suggests one good way to find useful tests is to follow threads of real users trying to do real work (Whittaker 2002). If you are utterly random and poking without purpose, you may be executing activities which your users would not realistically expect to produce a good result. Scott Barber likes the "cat on the keyboard test" as an example of a test in this realm. It's better to get teams engaged with interesting, challenging, and even confusing bugs related to critical features which users will use.

Here is a list of different schemes and tools to use for generating and performing robustness tests:

- Try some exploratory testing (as described by Cem Kaner and James Bach)
Starting with a charter and a time box, explore the application looking for opportunities to challenge the robustness of the system tied to the charter. Jot down notes for off-charter things to explore later. (For an overview, read "*Exploratory Testing Explained*" by Bach, 2003)
- Research historical data (for existing systems)
Study internal bug reports and externally reported user issues, logs of known failures, and other distressing metrics to discover patterns and trends of instability, and then build tests to evaluate those problems on your current project.
- Error guessing (ISTQB 2007)
Tapping the knowledge and experience of people (architects, developers, testers, subject matter experts, consultants, etc.) to perform analysis and anticipate what defects might be present in the component or system under test, and then design tests specifically to try and expose them.

Consider using Failure Modes and Effects Analysis (FMEA) or a fishbone/Ishikawa diagram to break down failures into potential root causes. Each root cause is a possible candidate for a test. I illustrate an informal way to use these schemes to find tests quickly in a previous conference presentation (Haynes 2008) named “Using Failure Modes to Power Up Your Testing.”

- Think like a software assassin (Hugh Thompson’s dream job title)
Go on a bug hunting mission by selecting targets (areas of the system that could be fragile or unstable) and use a pre-planned set of weapons (generic tests). Repeating operations or entering a large string of data for input are a couple of examples. Here are some resources that contain ideas for generic tests (please note: many ideas are duplicated because they are truly common):
 - Bug Taxonomies: Use Them to Generate Better Tests (Vijayaraghavan and Kaner 2003), contains a comprehensive list of taxonomies for various types of testing, but also outlines guidance for creating tests using taxonomies. Great article for getting started.
 - The book “How to Break Software” (Whittaker 2002) is written in tutorial fashion providing an outline for using a set of generic tests called *attacks*. The methodology targets common flaws in software relating to capabilities like input handling and computation, and explores both external (17 attacks for graphical user interfaces) and internal interfaces. Additional attack ideas can be found in the other books in the series:
 - “How to Break Software Security” (Whittaker and Thompson 2003)
 - “How to Break Web Software” (Andrews and Whittaker 2006)
 - The Test Heuristics Cheat Sheet on TestObsessed.com contains a heuristics list (example: Goldilocks – too big, too small, just right), data attacks and web tests, testing frameworks and tidbits of tasty testing wisdom. (Hendrickson, Lyndsay and Emery 2006)
 - Appendix A: Robustness Test Summary from the Robustness Testing of Software-Intensive Systems: Explanation and Guide (Cohen, Plakosh and Keeler 2005), contains a nicely formatted table of tests that includes general tests, GUI tests, network tests, database tests, disk I/O, registry and memory tests.
- Software Implemented fault injection (SWIFI, specifically runtime injection)
This scheme uses tools to rig stressful or unexpected scenarios for your software component or system. You still need to do the work of selecting relevant tests, but many tools have out of the box faults you can apply easily. In the book “How to Break Software,” Whittaker outlines a methodology for applying runtime injection in Appendix A (Whittaker 2002). Whittaker also provides a walkthrough of an older version of the currently available Holodeck fault simulation tool in Appendix B. This overview is a good introduction to the idea of using SWIFI. For additional information and tool references, please see my favorite uncles Google and Wikipedia.
- Bug bash
Bug finding parties are great ideas for team building. I always say, “if you feed them, they will come.” I’ve worked on many projects where we hosted a bug bash at least once during development, and sometimes more often if we were really concerned about unknown unknowns. We would invite folks from all over the company regardless of role or experience. We just wanted people that were interested and curious to come and play. Incentives were sometimes offered to the person who “bugged” us the most, or whoever delivered the best bug of all. Here’s some variations on the theme:

- “Bring your child to work day” a.k.a. “Watch the VP of Marketing use it for the first time” (Scott Barber’s personal favorite)
- Developer daze: have the developers from your project, or a different project have a go at the system.
- Revenge of the technical support team.
- Alpha test: invite your “favorite” users to break the house down – at your house, of course.
- English majors UNLEASHED! Invite the technical writers and documentation team members to “be free” and try all those crazy things they’ve been dreaming about.
- Product manager’s road test. (This one just might be fun to observe.)

While a bug bash can be a lot of fun, it can also generate a lot of data to pore through. You’ll need to plan on doing some rigorous triage and analysis of the bugs submitted. Seventeen instances of the “oops, I sat on the keyboard” defect might not turn out to be all that interesting. Other than that, bring a camera and post pictures or videos for a hall of fame, hall of shame, or a thank you board, and have fun!

Conclusion

Robustness testing is a way to explore invalid, unexpected and stressful scenarios, observe the results and provide stakeholder teams with information to determine a course of action. The earlier in a project these types of issues can be addressed, the better. Late discovery can impact schedule deadlines and be very costly. Shipping significant robustness issues could potentially impact technical support and service groups, cause users to lose confidence, or damage corporate image. Remember, people don’t find what they aren’t looking for - software robustness needs to be a mission across the project team if it’s to succeed! Taking a proactive stance on robustness testing is a way to reduce significant risk on many types of projects.

This paper has explained the importance of including robustness as part of your testing program, described the consequences of not including it, provided you with some tools and ideas to get started doing robustness testing. The referenced materials are a great source of more information as your robustness testing program matures.

References

- Andrews, M. and J. Whittaker. 2006.
How to Break Web Software: Functional and Security Testing of Web Applications and Web Services
Addison-Wesley Professional
- Bach, J. 1999. *Heuristic Rick-Based Testing*. <http://www.satisfice.com/articles/hrbt.pdf>
- Bach, J. 2003. *Exploratory Testing Explained*: <http://www.satisfice.com/articles/et-article.pdf>
- Cohen, J., Plakosh, D. and K. Keeler. 2005
Robustness Testing of Software-Intensive Systems: Explanation and Guide
Technical Note: CMU/SEI-2005-TN-015
<http://www.sei.cmu.edu/library/abstracts/reports/05tn015.cfm>
Carnegie Mellon University and Software Engineering Institute
- Guckenheimer, S. and J. Perez. 2006.
Software Engineering with Microsoft Visual Studio Team System
Addison-Wesley Professional
- Haynes, D. 2008.
Using Failure Modes to Power Up Your Testing (PowerPoint presentation)
Presented at STARWest 2008, <http://www.perftestplus.com/presentations.htm>
PerfTestPlus, Inc.
- Hendrickson, E., Lynsday, J. and D. Emery. 2006.
Test Heuristics Cheat Sheet
<http://testobsessed.com/wordpress/wp-content/uploads/2007/02/testheuristicscheatsheetv1.pdf>
Quality Tree Software, Inc.
- ISTQB [International Software Testing Qualifications Board, 'Glossary Working Party']. 2007.
Standard Glossary of Terms Used in Software Testing, version 2.0
ISTQB
- Kaner, C., Pels D. and D.L. Pels. 1998.
Bad Software: What to Do When Software Fails
John Wiley & Sons
- Platt, D. 2007. *Why Software SUCKS ... and what you can do about it*. Addison-Wesley
- Vijayaraghavan, G. and C. Kaner. 2003.
Bug Taxonomies: Use Them to Generate Better Tests (article)
<http://www.kaner.com/pdfs/BugTaxonomies.pdf>
- Weinberg, G. 2008. *Perfect Software and other illusions about testing*. Dorset House Publishing
- Whittaker, J. 2002. *How to Break Software: A Practical Guide to Testing*. Addison Wesley
- Whittaker, J. and H. Thompson. 2003.
How to Break Software Security: Effective Techniques for Security Testing
Addison Wesley

Half-Baked Ideas for Rapid Test Management

Jon Bach

jbtestpilot@hotmail.com

Abstract

As a test manager, there are stakeholders to report to and direct reports to collect reports from. There are processes and practices, principles and procedures, politics and protocols – all of which might bind you in some way from being creative. With all of that, “Moving Quality Forward” might mean you have to take some risk and break some rules.

This paper is about my experience as a test manager for one year at LexisNexis, leading 15 people on 4 projects. It’s about how I tried to find time to make the projects better by making my people better despite having no time for formal training.

It’s about the ideas I tried as a Rapid Test Manager focusing on the development of testing skill in each tester while testing was structured using heuristics and just-in-time documentation, rather than through detailed procedures and test case management systems. It’s about how I tried to move quality forward and what happened next.

Biography

I graduated from college as a journalist, but never pursued a career in it. I worked as a fireman, dishwasher, bookstore clerk, and even wrote a book about me and my Dad. But in 1996, I tried testing. At first, I didn't believe I could be very useful as a tester, because I wasn't a programmer.

My brother James Bach, who had been testing for 14 years at the time, said testing was about learning, but I didn't believe him. He said I needed to embrace complexity, and at that, I gave him no argument. He said my success as a tester would be determined by the speed and enthusiasm with which I learned about the products being tested. He said I needed is to be fearless about learning. He made it sound like a noble quest, then he gave me something to test.

Fourteen years later, I'm still here because it's a lot like journalism. In that time, I've been a keynote speaker at international testing conferences, I've co-authored a Microsoft Patterns & Practices book on Acceptance Testing, and I've led 15 people on 4 projects simultaneously. What follows is about that most recent job.

Copyright 2009, Jon Bach

Published at the Pacific Northwest Software Quality Conference 2009

A New Hope

I landed at LexisNexis last year – ready for my biggest challenge – leading 15 people on 4 different projects. Here, I was going to conquer the world with new ways to test and lead testers. I was going to experiment and have a lot of fun finally having a chance to manage a big group of testers the way I wanted to.

One project was in maintenance mode. Two others were agile-like, with testers working in two-week and six week sprints respectively. The fourth project was waterfall, sporting a long release schedule and formal test cycles.

For my staff, I was their third test manager that year.

A few of the guys were new to the team, but most had been grinding for a while. So, they were tired, and no one had much training as software testers.

I wanted to introduce this team to Rapid Testing, which is a testing methodology James developed over the last 20 years, in conjunction with Cem Kaner, Michael Bolton, and a little help from me. Rapid Testing focuses on the development of testing skill in each tester and the testing itself is structured using heuristics and concise documentation, rather than through ponderously detailed procedures or test case management systems. It does not consist of a set of approved test techniques or templates. Rather, Rapid Testing encompasses any useful test technique. There are no best practices, only best practices *in context*.

To manage the team well, I felt that I needed to do a few things quickly: establish my credibility as a tester and a leader, learn the technology under test, learn how the testers worked, and establish a mechanism for understanding testing status each day.

Here are some of the things I came up with. I call them "half-baked ideas" because I had no idea how they'd work or how they'd be received. They were literally ideas of ingredients that I put into the oven of a software project in my attempt to develop (bake) them.

What follows is an account of how they came out of the oven.

Family Feud

The first thing I wanted to do, like a lot of managers, is get to know my team. I held meetings with my crew, one by one, to get a feel for what was on their minds and what they might be concerned about. Based on what I heard, I made up a survey and sent it around.

The main purpose was to be sure I was hearing and understanding the important issues that that were on the minds of the testers. The survey included open-ended questions such as "What are 3 aspects of your ideal manager?" and "What one project problem do you

wished was solved NOW?" It also included some questions that were more closed and somewhat less serious, such as "Name the most interesting project codename we've used." and "Name a customer that tends to be mentioned a lot in meetings."

I decided to send a mail with the results, then stopped. Is that the most creative way I could introduce myself to the team? By sending out a simple email of results? Too boring.

So what popped into my head was the "survey says" game show, Family Feud. A hundred people surveyed about everyday objects, events, and choices. Contestants have to guess the results of the survey. So, I put on a suit and tie and played emcee for the game.

Result: Success. After I revealed the results, I took off the tie and told them what I was going to do to follow up on their answers. I don't know if anybody got anything out of it, but almost a year later, everyone remembers it. It was also fun because 6 months later, I sent out the survey results to remind them what their answers were to see how they'd changed. I was pleased to discover that the one project problem everyone wished was solved now (the slowness of Team Foundation Server), was solved.

Open-Book Testing

When I was a tester on Microsoft Flight Simulator in 2002, I had an idea. What if I could create a set of questions for myself that I did NOT know the answers to? And what if I let those questions guide my exploration to find new bugs?

I asked the testers to create a set of quiz questions about each of the products they were testing. Then I answered the questions by working with those products. In some cases, I hooked up a projector and answered the questions in real-time while my staff watched me flounder. That way they could see me learning. It was basically an open-book test.

The exam they created had questions such as:

- What is the difference between tags, notes, and issues?
- How can you blank an entire database?
- Where can you change field properties?
- How can you tell who is in the database at the same time as you?

Open-book testing is a great way to teach new testers about a particular technology. You don't tell them what it is and how to use it. Instead, ask them questions that force them to find out for themselves. As testers learn, they also may find bugs.

Result: Success. By putting myself through that process, I seemed to earn credibility with the team. When new testers came on board, they used the open book questions to guide their exploration. They also seemed to like coming up with new questions to ask themselves.

Dawn Patrol

A Dawn Patrol is a group testing event held before normal working hours – 6 AM, to be exact. We did one a week for four weeks. We picked a product to test and each tester tried to find bugs in it. There was a lot of chatter and questions going back and forth.

My role was to take notes. My screen was projected on a wall so they could see what I wrote. Apart from observing how they tested, I was introducing them to the idea of note-taking and responding to scrutiny, asking them why and how they did certain tests.

Managing Rapid Testing requires a very hands-on approach, at least some of the time. I think testing is best managed by getting directly involved in the work. I wanted my testers to know that I was a serious tester. I also needed to model for them the behavior I expected from a professional tester, so I got involved in the testing, too – right then and there, but also because I needed to learn all I could about the products they were testing.

Result: We never did them again. I'm not sure why. I got too busy, no one took the lead, it was too early in the morning, maybe all of these things. But it did build teamwork that lasted for my tenure there. I established myself as both a learner and a teacher, and showed how exploration was something they already knew how to do, but they could get better at it if they practiced.

This is an example of the "Training Wheels" effect in process improvement. A team that tries a new process may later drop it because they have learned something or changed in some way that makes the process no longer necessary. That's how training wheels work. Some organizations cling to procedures and paperwork that long ago lost relevance. But, in Rapid Testing, the point is to regularly ask ourselves if a simpler process might not work better.

Status Reporting Experiments

I had my staff send me written status reports at the end of the day so I could keep track of what they were all doing. They initially helped me get a feel for the rhythm of work on each of the test projects. The reports consisted of two sections: highlights and blocking issues. But as I gained a feel for the team, I stopped reading them and began to manage more by walking around. I did not tell them to stop producing them, however, because I wanted an archive. I wanted the option to read them weeks later, like cracking open a black box if there were to be a crash.

I also experimented with getting status by having the testers post index cards with testing tasks on their cube walls. This was an Agile-style dashboard so they didn't have to be interrupted when I came by their cube. I had testers post index cards on their walls showing the testing tasks in which they were currently engaged. That helped me understand the status of the team at any moment, just by walking around. But the idea

failed during the first Snow Day in December 2008 when LexisNexis wound up closing for the day. As people worked from home, I had no way to know their status.

But luckily, one of the testers realized that ScrumWorksPro would enable them to do the same thing online. So, now we use that. In other words, we found it was a worthwhile process, we just took it virtual.

Color-Aided Test Design

Some half-baked ideas came from the testers. Well, McDonald's, sort of.

McDonald's was making their yearly Monopoly gamepiece promotion, and I was collecting the pieces and posting them on my cube wall. Knowing I love analogies from life about testing, one of my testers dropped by to talk about rapid test design and he challenged me to say what Monopoly had to do with that. Thinking fast, I suggested that the different colored properties in Monopoly could be compared to different levels of test importance – Baltic for low priority, Boardwalk for high.

I then challenged him to do something with that idea. He decided to try color-coding his testing spreadsheet in the same way the Monopoly colors were represented on the board so that we could see, at a glance, the different kinds of tests on it.

But it turns out he didn't have an interest in it as much as he thought. He said it took a little too much work, and I believed him, though I didn't sit down with him to analyze why. He said it was hard to keep up with every new test he created and put in the spreadsheet. It was also hard to see the colors in a huge spreadsheet at a glance, so the purpose was moot.

Maybe someday I'll try it myself, perhaps on an Agile project with a Big Visible Chart of stories with colors denoting priority?

Testing Hybrids

Imagine a test case – a set of steps you follow to get an expected result. Now imagine a scenario – a vague mission statement or workflow that's up to you to invent to achieve an expected result. Hybrids were partially documented test procedures -- not detailed step-by-step procedures, but also not a typical exploratory testing charter. It is a checklist, but it is a checklist of ideas of things to see and do, like a trip itinerary.

It's another idea that came from the team. I was telling them that we could test better and faster if we ignored thick test procedure documentation. But one of the testers felt that the hybrid approach could give me the structure I wanted while cutting way down on paperwork.

Rapid Testing works well this way. By asking "What testing is needed, here and now? How do we prepare ourselves for testing that we'll need to do tomorrow? Are we

delivering the information our clients need?" again and again, every day, perhaps several times a day. In Rapid Testing, you are either a test leader, or you are training to become a test leader. I believe that testing is a challenging intellectual investigation, and Rapid Testing is about getting the most out of people to meet that challenge.

That's why encouraging testers to try experiments within (or without) a test process is important. They try things, they learn from them, and they come to own the process for themselves. This gives them confidence and makes them more committed to their work.

Here are some other ideas, quickly baked and out of the oven:

Lightning Talks – Book your team in a conference room for one hour with a projector and a timer. But three days prior, give them time to volunteer to deliver a 5-minute talk on anything project-related that they think the team could benefit from knowing – a tool demo (like using the free Windows Media Encoder to capture video) , a trick (like pressing F5 in Notepad to imprint an instant timestamp), or a website (like the wonderfully useful www.testingreflections.com).

Tester Show and Tell – this is like the lightning talks, only longer. This can happen in email, too. Ask your staff their favorite things to show off. What are their favorite Windows shortcuts (like holding down the Windows logo key and pressing E to launch Windows Explorer), what's their favorite testing website, what tools do they have loaded on their machine that are really useful? What the best bug they ever found on their current project? Give them a chance to show off or to simply recommend something and you may find a lot of "a ha" moments. That's what happened to me. I learned about Windows Media Encoder and <Windows key>+L to lock Windows, and in exchange, I showed them PerlClip.

Using WME and VLC for videos – Give them an assignment to record a 5-minute video of a test they did or something useful to the team. They might end up using it to record a short screencast of the next bug they find and attach the WMV file to the bug report. Both are free tools, and the ensuing recording can be put in a training archive. It gives your testers a chance to practice story-telling and reporting, as well as refine their exploration techniques. LiveMeeting also allows screencasts to be recorded and stored for future playback.

Dogfooding – (comes from "eating our own dogfood" – a way to use the product you're testing as if you were a customer yourself) -- Is there a way you can use your product you're testing to accomplish a task on your project? For example, if you're shipping a database-based product, can you test it by creating a database of all of your tests? If it's a web-based wiki dashboard app of some kind, can you use it to track project status updates? One of my testers used his product CaseMap – a tool for helping lawyers organize facts, witnesses, and evidence -- to organize his test cases and found

Conclusion

When I join a test team, I have my favorite ways of working, like everybody. I am biased toward exploratory testing and session-based test management. I'm skeptical of thick documentation, grandiose test automation, and heavy process. Even Agile projects don't seem agile enough for me. There is a lot of discipline required and not all technical professionals can work like that.

As a test manager, I think it's important to float ideas out there without over-testing them first. I take a service approach to leadership, and I'm excited to solve a problem fast, even though it's not all the way thought through. I'd rather trade time for the fun of experimentation. The risk, of course, is that haste makes waste. A half-baked idea can come out of the oven doughy and mushy. But it can also be "good enough" for its purpose.

Good enough does not mean mediocre. It means, "stopping the search for alternatives due to the cost of finding a more optimal solution." The word "good" means quality (or value) and the word "enough" means stop. If further time spent trying to improve an idea is too costly, take it out of the oven the way it is and see what happens.

In other words, look at management situations not in terms of a machine that must operate in some rigorous and inhuman way, nor a math problem to be cleverly solved. See yourself as a tester trying to move the quality of your team forward in little ways. Your half-baked idea may just be the training wheels you needed to get past the given hurdle or make your point.

BEST PRACTICES FOR SECURITY TESTING

TOP 10 RECOMMENDED PRACTICES

Aarti Agarwal

McAfee Software P. Ltd (Enterprise Security Team), Bangalore, India
Tel +1 9980951530 • Email Aarti.Agarwal@gmail.com

Abstract

This White Paper is intended to outline and define the process, procedures and methods that should be used to evaluate the product from a security perspective. The “Top 10” recommended security practices have been prioritized for building confidence in your product, focusing on the most critical areas every enterprise needs to adopt.

About The Author

Aarti Agarwal is a Software Quality Assurance Engineer in McAfee’s Enterprise Security team. She has an experience in Testing and Quality Assurance with a special focus on security testing of client and server applications.

In her present role, she is the research and analysis owner for security testing of McAfee’s Host Intrusion Prevention System that preserves desktops’ and servers’ integrity with signature and behavioral protection from attacks. Her role involves identification of architectural risks, design and implementation risks, information gathering, user interface attacks, file system attacks, design attacks, implementation attacks, test planning, threat model preparation, research on security tools, penetration testing, execution and analysis of results and report metrics preparation.

She has also been associated with Accenture Services Pvt. Ltd, where she gained expertise in Data Warehouse quality assurance for a Financial Organization project.

Copyright Aarti Agarwal August 15, 2009

Published at the Pacific Northwest Software Quality Conference 2009

From a technical and project management perspective, security test activities are primarily performed to validate a system's conformance to security requirements and to identify potential security vulnerabilities within the system. From a business perspective, security test activities are often conducted to reduce overall project costs, protect an organization's reputation or brand, reduce litigation expenses, or conform to regulatory requirements. Identifying and addressing software security vulnerabilities prior to product deployment assists in accomplishing these business goals. Security issues are among the highest concerns to many organizations. Despite this fact, security testing is often the least understood and least defined task.

There are two major aspects of security testing: testing security functionality to ensure that it works and testing the subsystem in light of malicious attack. Security testing is performed by probing undocumented assumptions and areas of particular complexity to determine how a program can be broken. In addition to demonstrating the presence of vulnerabilities, security tests can also assist in uncovering symptoms that suggest vulnerabilities might exist.

This White Paper is intended to outline and define the process, procedures and methods that should be used to evaluate the product from a security perspective. Here, I'll prioritize the "Top 10" recommended security practices for building confidence in your product. These recommendations can be used as a guideline for security testing of system as well as web based products. These guidelines aim to bridge the gap in knowledge and process between vulnerability researchers and quality assurance professionals by bringing together software security and software testing expertise.

1 Changing the Old Paradigm: Creating a better, more secure application development process

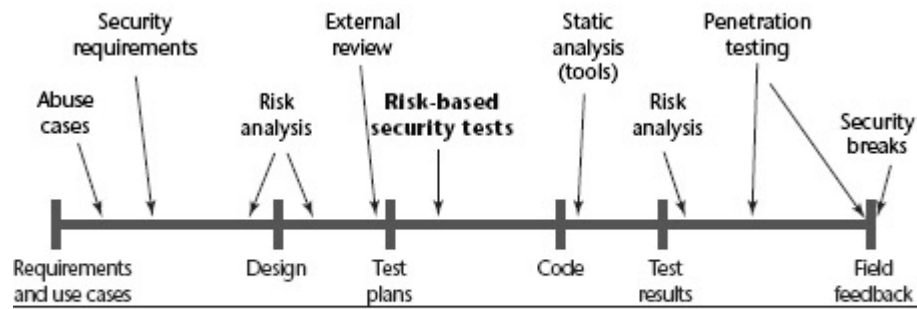
Software security testing is very different from software functionality testing and hence, simple "black-box" testing is not enough. Security best practices should be integrated into the software development lifecycle because they reduce overall costs by increasing efficiency and improve customer satisfaction.

Testing for security vulnerabilities requires a full understanding of how the application is designed, and the ability to creatively think about what is going on inside the application as it is operating. Simple "black-box" testing is not enough.

In order to break this traditional approach, we must fundamentally change the way that we approach application security. We must integrate security throughout the Software Development Life Cycle (SDLC), not just hastily add it to the end.

Security testing in the Software Life Cycle

Integrating security best practices into the software development lifecycle actually reduces overall costs by increasing efficiency and also boosts customer satisfaction. Since risk analysis is an ongoing and fractal process throughout software development, new information is always becoming available for the test plan and test planning becomes an ongoing process as well. The figure below shows the security practices in different phases of the software development life cycle:



Integration of security practices through SDLC

The Start of the Software Life Cycle

The preparations for security testing can start even before the planned software system has definite requirements and before a risk analysis has begun. *For example, past experience with similar systems can provide a wealth of information about how attackers have typically tried to subvert similar systems.*

More generally, the initiation phase makes it possible to start a preliminary risk analysis, asking what environment the software will be subjected to, what its security needs are, and what impact a breach of security might have. If risk analysis starts early, it will provide an early focus for the test planning and security-oriented approach when defining requirements.

Security Testing in the Requirements and Design Phases

The process of software development starts by gathering requirements and developing use cases. In this phase, *test planning* focuses on outlining how each requirement will be tested. Mapping out the elements of the security test plan should begin in the requirements phase of the development life cycle in order to avoid surprises later on. Details on the test plan are to follow later in this paper.

It is also useful for a more detailed risk analysis to begin during the requirements phase. Security analysis often uncovers severe security risks that were not anticipated in the requirements process. The risks identified during this phase may lead to additional requirements that call for features to mitigate those risks. The software development process goes more smoothly if mitigations are defined early in the life cycle, when they can be more easily implemented.

Security Testing in the Test/Coding Phase

Functional security testing generally begins as soon as there is software available to test. A test plan should be in place at the start of the coding phase along with the necessary infrastructure and personnel.

This document does not attempt to catalog every possible testing activity. Instead, it will discuss several broader activities and the role of security testing in each. The activities discussed are as below:

Unit Testing

Unit testing is usually the first stage of testing that a software code goes through. It is where individual classes, methods, functions, or other relatively small components are tested.

When evaluating potential threats to a software component, one should keep in mind that many attacks have two stages: one stage where a remote attacker gains initial access to the machine, and one stage where the attacker gains control of the machine after gaining access. *For example, an attacker who subverts a web server will initially have only the privileges of that web server, but the attacker can now use a number of local attack strategies such as corrupting system resources and running privileged programs in corrupt environments.* Therefore, defense in

depth demands that local threats be addressed on systems that are assumed not to have malicious users, and even on machines that supposedly have no human users at all. Also, assumptions a component makes about its environment should be tested for vulnerability to exploits.

Testing Libraries and Executable Files

In many development projects, unit testing is closely followed by a testing effort that focuses on libraries and executable files.

A greater level of testing expertise is needed for this testing as the testers need to be up to date on the latest vulnerabilities and exploits¹. The test environment can also be complex, encompassing databases, stubs for components that are not yet written, and complex test drivers used to set up and tear down individual test cases. It is useful during functional testing to carry out *code coverage analysis* using a code coverage tool. This helps identify program parts not executed by functional testing. These program parts may include functions, statements, branches, conditions, etc. Such an analysis helps to

- focus and guide testing priorities
- assess the thoroughness of testing
- identify code not executed by tests
- check adequacy of regression test suites
- keep test suites in sync with code changes

Error handling routines are difficult to cover during testing, and they are also notorious for introducing vulnerabilities. Good coding practices can help reduce the risks posed by error handlers, but it may still be useful to have testware² that simulates error conditions during testing in order to exercise error handlers in a dynamic environment.

Libraries also need special attention in security testing keeping in mind: just because a library function is protected by other components in the current design does not mean that it will always be protected in the future. *For example, a buffer overflow³ in a particular library function may seem to pose little risk because attackers cannot control any of the data processed by that function, but in the future the function might be reused in a way that makes it accessible to outside attackers.*

Furthermore, libraries may be reused in future software development projects, even if this was not planned during the design of the current system. This creates additional problems.

1. People who developed the library code might not be available later, and the code may not be well understood anymore
2. Vulnerabilities in the library will have a greater negative impact if it is reused in many systems.
3. If the library is used widely, malicious hackers might have exploits already at hand.

An example of a vulnerable library function is the `strcpy()` (string copy) function in the standard C library, which is susceptible to buffer overflows. For many years, calls to `strcpy()` and similarly vulnerable functions were one of the chief causes of software vulnerabilities in deployed software. By the time `strcpy()`'s vulnerability was discovered, the function was in such widespread use that removing it from the standard C library was not feasible. The story of `strcpy()` contains a moral about the value of catching vulnerabilities as soon as possible and the expense of trying to address vulnerabilities late in the life cycle.

Integration Testing

Integration testing focuses on a collection of subsystems, which may contain many executable components. There

¹ A few recommendations for information on the latest security vulnerabilities: <http://www.securityfocus.com>, <http://www.microsoft.com/technet/security>

² Testware is an umbrella term for all utilities and application software that serve in combination for testing a software package but not necessarily contribute to operational purposes. Testware is produced by both verification and validation testing methods.

³ A buffer overflow occurs when a program or process tries to store more data in a data storage area than it was intended to hold. The extra information can overflow into the runtime stack, which contains control information such as function return addresses and error handlers.

are numerous software bugs that appear only because of the way components interact, and this is true for security bugs as well as traditional ones. Integration errors are often the result of:

- One subsystem making unjustified assumptions about other subsystems
- Failure to properly check input values. During security testing, it is especially important to determine what data can and cannot be influenced by a potential attacker.
- Error handlers because of unusual control and data-flow patterns during error handling. Error handling is one more form of component interaction, and it must be addressed during integration testing.

System Testing

In this stage, the complete system is the artifact that will actually be attacked. *Penetration testing* involves manually or automatically trying to mimic an attacker's actions and checking if any tested scenarios result in security breaches. Penetration testing makes the most sense here, because any vulnerability it uncovers will be a real vulnerability.

Two levels of penetration testing are recommended:

- To determine whether the application is vulnerable to **common attacks**, use penetration testing, which simulates the types of attacks that could be launched on any application.
- To determine whether the application is vulnerable to **unique attacks** (such as attacks on application logic), design and execute penetration tests that attempt to subvert your application's unique security mechanisms. This is critical to verify that an application can resist the application logic attacks that are now becoming increasingly common.

Stress testing is also relevant to security because software performs differently when under stress and many vulnerabilities come out because of conditions that developers were not expecting. *For example, when one component is disabled due to insufficient resources, other components may compensate in insecure ways.* An executable that crashes altogether may leave sensitive information in places that are accessible to attackers. Attackers might be able to spoof subsystems that are slow or disabled, and race conditions might become easier to exploit. Stress testing may also exercise error handlers. Unusual behavior during stress testing might also signal the presence of unsuspected vulnerabilities. *Metasploit* is a good tool in this space.

The Operational Phase

The operational phase of the software life cycle begins when the software is deployed. Often, the software is no longer in the hands of the developing organization or the original testers. Traditionally, beta testing is associated with the early operational phase, and it has its counterpart in the security arena, since white-hat hackers may examine the software in search of vulnerabilities.

However, *beta testing* is not the last word because the software system may *become* vulnerable after deployment. This can be caused by configuration errors or unforeseen factors in the operational environment. Assumptions that were true about those components during development may no longer be true after new versions are deployed. It may also be that only some components are updated while others are not, creating a *mélange* of software versions whose exact composition cannot be foreseen in any development setting. This makes the vulnerabilities of the composite system unforeseeable as well.

The risk profile of the system might also change over time, and this creates a need for continuing *security audits* on systems deployed in the field. This can happen when the system is used in ways that were not foreseen during development, or when the relative importance of different assets grows or diminishes. *For example, an organization might unknowingly protect critical information with encryption methods that are no longer considered secure.*

For new vulnerabilities that are uncovered in existing software versions. A responsible development organization will release a patch, but software users might not apply the patch. This creates a drastic shift in the system's risk profile.

A related issue is that *encryption techniques* used by a software system can become obsolete, either because increasing computational power makes it possible to crack encryption keys or because researchers have discovered ways of breaking encryption schemes previously thought to be secure. These issues create the need for *security audits* in deployed systems.

2 Information Gathering: Software Components and Their Environment

A thorough understanding of the software and its environment is necessary to evaluate the attack surface. Information can be gathered by Runtime Analysis, Application footprinting, OS footprinting, Environmental Interactions and with the help of some appropriate tools.

A thorough understanding of the product's design is necessary. You need to know where all the entry points are so that you can understand and diagram the product's *attack surface*.

Evaluating the attack surface lets you target the areas where an attacker might find an entry point. To understand the high-risk areas of a business application, you also need to know the valuable functions that the program performs and its assets/resources. Typically this requires reviewing the program's design documents, designing a threat model, including data flow diagrams if they exist, and interviewing the program's designers or architects.

Additional attack surface information can be gathered by **Runtime Analysis**—executing the program and analyzing it with debugging and diagnostic tools to create a block diagram of the system, sketching the major components and the major data flows between them. Of special importance is the data that comes from outside the program that must be considered untrusted. These are the inputs to the system where an attacker can strike. Here are some external data flows:

- Network I/O
- Remote procedure calls (RPCs)
- Querying external systems: domain name system (DNS), database lookups, Lightweight Directory Access Protocol (LDAP)
- File I/O
- Registry
- Named pipes, mutexes, shared memory, any OS object
- Windows messages
- Other operating system calls

Application footprinting

It is the process of discovering what system calls and system objects an application uses. It lets you know how an application receives input from its environment via operating system calls. It also helps to find what system objects the application uses, such as

- Network ports
- Files
- Registry keys

You can use the application footprinting process to validate and add to the information gathered from the runtime analysis.

Windows footprinting

Process Explorer shows you what DLLs and handles a process has opened. As the program executes, handles are created and destroyed. So *Process Explorer* is good at discovering the system objects that the process uses for long periods of time. For shorter-lived objects or object accesses, you need to use other tools.

Process Explorer can view the following handles:

- | | | | |
|-------------|--------------|-----------------|----------|
| • Desktop | • Process | • WindowStation | • Timer |
| • Directory | • Section | • Port | • Thread |
| • Event | • Semaphore | • Token | • Key |
| • File | • KeyedEvent | • SymbolicLink | • Mutant |

Other useful tools for Windows footprinting are *Filemon*, *Regmon* and *Procmon* from the Windows Sysinternals Suite. These tools intercept calls to the Windows file and Registry APIs. *Procmon* also shows real-time file system, registry and process/thread activity.

Unix footprinting

strace, *ktrace*, and *truss* are debugging tools that print a trace of all the system calls made by an application. *strace* is available on Linux. *ktrace* is available on FreeBSD, NetBSD, OpenBSD, and OS X. *truss* is available on Solaris.

It is important to understand the software itself, but it is also important to understand the **environment**. A software module may interact with the user, the file system, and the system memory in fairly obvious ways, but behind the scenes many more interactions may be taking place without the user's knowledge.

Two types of environmental interactions have to be considered, namely, corruption of the software module by the environment and corruption of the environment by the software module. Some of the environmental interactions to be considered include:

- interactions with the user
- interactions with the file system
- interactions with memory
- interactions with the operating system via system calls
- interactions with other components of the same software system
- interactions with dynamically linked libraries
- interactions with other software via APIs
- interactions with other software via interprocess communication
- interactions with global data such as environment variables and the registry
- interactions with the network
- interactions with physical devices
- dependencies on the initial environment

The biggest challenge is to avoid overlooking an interaction altogether. Still, it is appropriate to prioritize these interactions according to how realistic it is to view them as part of an attack.

Some good tools in this space are *Log Parser* for information gathering, *Application Verifier* for runtime analysis, and *Total Uninstall* for monitoring registry and file system changes.

3 Threat evaluation with Data Flow Diagrams

Building and analyzing a DFD helps in understanding how the system works and the threats that your system faces. The STRIDE model is a good way to get started in this space.

A data flow diagram (DFD) is a block diagram. A DFD with security-specific annotations will help to describe how data enters, traverses and leave the system. It shows data sources and destinations, relevant processes that data goes through and trust boundaries in the system.

Be particularly sensitive to data that flows into your control from *external sources*, including: script from web pages, files read from the local computer, user input, and data read from the registry. One good strategy is to consider every kind of threat possible at each data entry point. Microsoft's **STRIDE⁴** model provides good help in evaluating the threats.

4 Prioritizing Security Testing with Threat Modelling

Threat modelling is an iterative procedure for prioritizing security testing. It involves identifying threat paths, threats, vulnerabilities, ranking the risk associated with vulnerability and determining exploitability. Tools can assist in defining strategies for dealing with security issues.

Focus testing on areas where difficulty of attack is least and the impact is highest.

—Chris Wysopal

It is necessary to prioritize the tests to be performed to maximize the quantity and severity of the security flaws uncovered given a fixed amount of time and resources. A technique for prioritizing security testing is *threat modelling*. It is an iterative procedure for optimizing network security by identifying objectives and vulnerabilities, and then defining countermeasures to prevent, or mitigate the effects of, threats to the system.

The threat-modelling process has four main steps:

Identifying Threat Paths

The first stage of the process is to identify the highest-level risks and protection mechanisms for the application you will test. First, overall security strengths of the application platform and implementation language are noted because these will be relevant throughout the threat-modelling process. Next, you need to identify the different user access categories. The table below shows sample user access categories ranked from highest-risk to lowest-risk.

⁴ Spoofing identity, Tampering with data, Repudiation, Information disclosure, Denial of service and Elevation of privilege.

Access Categories	Risk Access Category
Very high	Anonymous remote user
High	Authenticated remote user with file manipulation capability
Medium	Authenticated remote user
Low	Local user with execute privileges
Very low	Administrative local user

Most client/server applications have similar user access categories. Every application needs security testing performed on the threat paths that anonymous and authenticated remote users can access. The next step in identifying the threat paths starts with a DFD as explained earlier.

Identifying Threats

Starting with the highest-risk threat path, the processing performed along the path is analyzed. The following is a list of questions to ask for each processing component along the path:

- What processing does the component perform?
- How does it determine identity?
- Does it trust data or other components?
- What data does it modify?
- What external connections does it have?

Then high-risk activities need to be marked along the path:

- Data parsing
- File access
- Database access
- Network access
- Authentication
- Authorization
- Synchronization or session management
- Handling private data
- Spawning of processes

Use this information to write out the threats along each threat path. This list of threats will be used to drive the vulnerability finding process.

Identifying Vulnerabilities

A threat becomes a vulnerability when the designers fail to build any security features into the application that mitigate that threat. Some of the security mitigations to look for are *data validation testing*, *resource monitoring*, and *access control* for critical functions.

The vulnerability hunt can branch in several directions at this point: detailed security design review, security code review, or security testing. The security design review is best at finding design level vulnerabilities. The security code review and security testing are best at finding coding errors where the programmer did not follow secure coding practices. These three methods of security analysis each benefit from the prioritization done during the threat-modelling procedure.

Ranking the Risk Associated with a Vulnerability

A technique for ranking a threat's severity is to use the DREAD model (introduced by Michael Howard and David Leblanc). You rank a threat using DREAD as follows:

- Damage potential - The extent of the damage if vulnerability is exploited.
- Reproducibility - How often an attempt at exploiting vulnerability works.
- Exploitability - How much effort is required? Is authentication required?
 - authentication required,
 - no authentication but hard-to-determine knowledge required,
 - no authentication and no special knowledge required.

- Affected users - How widespread could the exploit become?
 - only rare configurations,
 - a common case,
 - the default for most users.
- Discoverability (you might not want to use this one if you believe all vulnerabilities are eventually discovered)

Determining Exploitability

When software is in development, it is typically easier to just fix an issue that looks likely to be exploitable than to take the time to determine if it actually is.

Threat Analysis & Modelling (TAM) Tool

Microsoft's TAM tool helps build a picture of threats the systems may face and assists in defining strategies for dealing with security issues. TAM also aids in collectively increasing the security awareness of a team and customers.

Create Test Cases based on Threats Identified in the Threat Model

For each threat identified in your threat models, you should create a test case. The test cases should confirm that the identified threat has been mitigated by the design or implementation of your control.

5 Identifying Vulnerabilities in Source Code

Well-tested code that includes security tests results in an end product that is more robust, easier to maintain and more secure. This can be implemented through security code reviews, source code analysis (can be automated) and observing the behavior of running applications.

The best time to begin looking for security vulnerabilities during development is when writing code. There are coding techniques that produce functional results, yet have unresolved errors in the application and allow intruders to gain unauthorized access.

A few ways to detect vulnerabilities and violations of secure coding practices are through:

- Security code reviews
- Automating the code review process with source code analysis
- Observing the behavior of the running application to discover poor practices such as unencrypted passwords being transferred over the network, storing weakly encrypted passwords in the system, database connections that are opened but not closed and even unhandled exceptions within the application.

Here are a few pointers to keep in mind while identifying vulnerabilities in source code (C/C++):

- For memory management, unbound API functions are particularly dangerous like:
 - strcpy()/tcscopy(), strcat()/tcscat(), the *sprintf()/_tsprintf() family of functions, the *scanf() family of functions, Gets(), strncpy()/_tcsncpy() – these do not guarantee NULL termination
 - strncat()/_tcsncat() - deceptive size argument. Size is not the size of the buffer, it is the size (amount of data) left in the buffer-1 (it does NOT account for NULL byte)
 - (v)snprintf()/_tsnprintf() - subject to return value misinterpretation.
- In pointer arithmetic, the most common error is "off-by-one" mistakes which results in a single byte being written out of bounds.

- In case of (v)sprintf(), using format specifiers like "%n" might extend the amount of output data you can write, thus triggering a buffer overflow.
- To avoid buffer overflows, developers usually enforce a max limit on the input and do not copy more than that. This methodology has the potential side-effect of data being lost.
- In Program Flow, the size parameter usually indicates the size of a buffer in wide characters, not bytes. Incorrectly supplying a size in bytes is easy to do and can result in memory corruption. E.g.: MultiByteToWideChar(), wcsncpy(), wcsncat().

Automating this type of analysis can be faster and more comprehensive than a manual effort and is possible, using a product that observes both the internal calls and data transfers within the application, the operating system and software environment in which it operates.

6 Testing with Known Intrusions

Test the product with the known intrusions, as no security testing regime is complete until you actually try to break into the running application. Automating attack simulation is a good approach.

Even if you have a degree of assurance that the application was developed with good coding practices and security in mind, inadvertent errors or unforeseen consequences of specific practices can create security weaknesses that is not apparent using other types of analyses. So, test the product with the known intrusions.

A better approach is to automate attack simulation, a process whereby software pretends to attack the application with known paths of intrusion. The value of attack simulation is that it verifies the application cannot be breached by known means. If it can, then the problem can be fixed and verified before the application goes into production. Standard tools like *Metasploit* can be used to test the software with known and current intrusions.

7 Creating a Security Test Plan

The Security Test Plan should incorporate a high-level outline of which artifacts are to be tested and what methodologies are to be used, including a general description of the tests themselves, prerequisites, setup, execution, and a description of what to look for in the test results.

The main purpose of a test plan is to organize the testing process, including security testing. It outlines which components of the software system are to be tested and what test procedure is to be used on each one. Often, the test plan has to account for the fact that time and budget constraints prohibit testing every component of a software system; risk analysis is one way of prioritizing the tests.

Test planning is an ongoing process. In creating the test plan, inter-component dependencies must be taken into account so that the potential need for retesting is minimized. The planning process also includes validation of the test environment and the test data, which should be verified during test design and execution.

Within each cycle, the test case specification should map out the *test cases* that should run during the test execution process. A test case typically includes information on the preconditions and post conditions of the test, how the test will be set up, how it will be torn down, and how the output of the test will be evaluated. The test case also defines a test condition, which is the actual state of affairs that is going to be tested. The process of actually devising test conditions is a challenging one for security testing. The test requirements, test case specifications and test procedures can be separate documents. It aids in maintenance, since the test plan changes less than the test specifications, which change less than the test cases.

A typical test plan that includes security considerations might contain the following elements:

- Purpose
- Software Under Test Overview
 - Software and Components
 - Test Boundaries
 - Test Limitations
- Risk Management
 - Synopsis of Risk Analysis
 - Mitigation Plan
 - Contingency Plan (Back-up plans or Worst-case scenario plans)
- Test Strategy
 - Assumptions
 - Test Approach
 - Items Not To Be Tested
- High Level Test Requirements
 - Functionality Test Requirements
 - Security Test Requirements
 - Installation/Configuration Test Requirements
 - Stress and Load Test Requirements
 - User Documentation
 - Personnel Requirements
 - Facility and Hardware Requirements
- Test Environment
- Test Case Specifications
 - Unique Test Identifier
 - Requirement Traceability (what requirement number from requirement document does test case validate)
 - Input Specifications
 - Output Specifications/Expected Results
 - Environmental Needs
 - Special Procedural Requirements
 - Dependencies Among Test Cases
- Test Automation and Testware
 - Testware Architecture
 - Test Tools
 - Testware
- Test Execution
 - Test Entry Criteria
 - QA Acceptance Tests
 - Regression Testing
 - Test Procedures(Special requirements, Procedure steps)
 - Test Schedule
 - Test Exit Criteria
- Test Management Plan
- Definitions and Acronyms

8

Choosing the Right Tool for Security Testing

Choosing an adequate tool is very important. Automated security testing is a comprehensive approach that spiders the entire application to identify security holes but can also produce false positives. Manually crawling a web application can be time consuming, but it also helps ensure that specific pages are tracked and analyzed.

The QA department will need application security testing software that can perform three types of testing—as a *non-authenticated user*, an *authenticated user* and an *administrative user*—to determine the vulnerabilities inherent in each user class. Additionally, the web application security tool should be able to perform crawling/spidering⁵ of your Web application.

Manual security testing allows a user to focus on specific pathways or tasks on a web site while the software follows silently behind, tracking the process. The program can then audit the particular path that the user has taken for security vulnerabilities and provide a report. Manually crawling an application can be time consuming, but it also helps ensure that specific pages are tracked and analyzed.

Automated security testing will spider the entire application by clicking every button and link, filling out data fields to identify the structure of the program, and then audit each page for vulnerabilities. It should do this from the outside in, reviewing each portion of the site the way an external hacker might, ideally from behind the scenes. On the down side, it can also produce false positives, and it may not be able to access all of your Web pages due to the way certain pages are coded.

Here is a special mention of a tool, *BackTrack*, which contains a free suite of open source security tools and is very logically structured according to the work flow of security professionals. Currently it provides a graphical environment with **more than 300 different up-to-date tools**. The major tool categories are:

- Information Gathering
- Network Mapping
- Vulnerability Identification
- Penetration
- Privilege Escalation
- Reverse Engineering
- Maintaining Access
- Radio Network Analysis
- VOIP & Telephony Analysis
- Digital Forensics
- Miscellaneous

⁵ A Web crawler is a computer program that browses the World Wide Web in a methodical, automated manner. This process is called Web crawling or spidering.

9 Relevant Metrics

Both the quantity and quality of testing needs to be measured to help make a quantifiable assessment about the efficiency of the testing performed on the software. The metrics that are useful include test requirements criteria, test coverage criteria, test case metrics, defect metrics and test case quality metrics.

Unfortunately, there are no industry-standard metrics for measuring test quantity and test quality. In general, test metrics are used as a means of determining when to stop testing and when to release the software to the customer. The test criteria to be considered and the rationale behind using these criteria are described below.

• **Requirements test criteria:** One of the purposes of testing is to demonstrate that the software functions as specified by the requirements. Thus every software requirement must be tested by at least one corresponding test case. By having traceability from test cases to functional requirements, one can determine the percentage of requirements tested to the total number of requirements. Thus, *test requirement metric* is defined as the ratio of requirements tested to the total number of requirements.

• **Test coverage criteria:** Testing should attempt to exercise as many "parts" of the software as possible, since code that is not exercised can potentially hide faults. Exercising code comprehensively involves covering *all* the execution paths through the software. Unfortunately, the number of such paths in even a small program can be astronomical, and hence it is not practical to execute all these paths. The next best thing is to exercise as many statements, branches, and conditions as possible. Thus, we can define the test coverage criteria such as statement coverage, branch coverage, etc. *Commercial off-the-shelf (COTS) coverage tools* can be used to monitor coverage levels as testing progresses.

• **Test case metrics:** This metric is useful when it is plotted with respect to time. Essentially, such a plot will show the total number of test cases created, how many have been exercised, passed and failed over time. These project-level metrics provide a snapshot of the progress of testing where the project is today and what direction it is likely to take tomorrow.

• **Defect metrics:** These metrics are very useful when plotted with respect to time. It is quite informative to plot cumulative defects found since the beginning of testing to the date of the report. The slope of this curve provides important information. If the slope of the curve is rising sharply, large numbers of defects are being discovered rapidly, and certainly testing efforts must be maintained, if not intensified. In contrast, if the slope is leveling off, it might mean fewer failures are being found. It might also mean that the quality of testing has decreased. Defect metrics also include measures of the following criteria as an information for analysis:

- total open, by priority
- show-stopper / stop-ship problems
- find rate, by severity (daily and 15-day rolling average)
- resolution type versus severity for all resolved problems (total and last 15 days)
- unreviewed problems, by priority
- unverified resolutions (QA backlog)
- reopen rate, over time (number of problems with resolutions that were rejected by QA, and thus reopened rather than concluded).

• **Test case quality metric:** Test cases "fail" when the application under test produces a result other than what is expected. This can happen due to several reasons, one being a true defect in the application. Other reasons could be a defect in the test case itself, a change in the application, or a change in the environment. Test

case quality metric can be defined as the ratio of test case failures resulting in a defect to the total number of test case failures.

The most important of these metrics are defect metrics. Defect metrics must be collected and carefully analyzed in the course of the project.

10 Reporting

All time-oriented metrics should be measured on more or less a daily scale. All problems found by anyone during any phase of testing must be logged in the database. Defect metrics are vital to the successful management of a high assurance test project and hence needs to be used properly.

All time-oriented metrics should be measured on a daily scale. These metrics should be automated so that they are updated on a daily basis. The reports should be available for the whole system and on the level of individual product areas.

The major reports to produce include Test case metrics, Defect metrics and Test case quality metrics (as detailed above in section 9). Others may be useful, depending on the circumstances

Problem Tracking

The problem database should be maintained primarily by the test team. All problems found by anyone during and after functional testing must be logged in the database. This protocol can be enforced easily by requiring that all changes made after code freeze be associated with a problem number. Prior to each build, the change list is then compared to the problem tracking system. Problems found by customer engineering, and ultimately the customer, are also important to track.

How To Use the Metrics

Defect metrics are vital to the successful management of a high assurance test project. Here are a few important points to keep in mind:

- Defect metrics should be used on more or less a daily basis, to review the status of the product, scan for risks, and guide the testing effort toward areas of greatest risk.
- The critical and severe open problem reports should be resolved and verified before the product is shipped.
- A find rate requirement (e.g., no defects found for a week) is not recommended for shipping the product. Reviewing each problem found and using that information to question the efficacy of the test effort and to re-analyze technical risk is recommended. Keeping the important problem find rate as high as possible throughout the project is an important goal of the test team.
- The reopen rate should start low (less than 1 out of 10 resolutions rejected) and get steadily lower over the course of the project.

Conclusion

The importance of security is no secret in the world of software today. The stakes are incredibly high and as products grow in complexity and sophistication, so has securing and operating software become more challenging and demanding.

The pressure and more importantly the need to build software products with improved security features to fulfill their economic promise and protect organizations against liability and loss is more than it has ever been.

Software security, today, needs a highly efficient methodology to plan, develop, test and maintain the softwares which are under constant attack by the hackers with malicious intent. My "Top 10 best practices" recommendations is an effort towards the development of one such methodology and focuses on some of the most critical areas that every enterprise needs to adopt. As the technology and thus security threats are constantly evolving, no solutions can promise a foolproof secure system but following some best practices like the ones mentioned here, we can definitely take a respectable stand against, and possibly even get ahead of, the cyber criminals who are already planning their next moves.

References

- MCGRAW, Gary: *Software Security* (Addison-Wesley; 2006).
- J. Whittaker and H. Thompson: *How to Break Software Security*, Addison-Wesley, 2003.
- <https://buildsecurityin.us-cert.gov/daisy/bsi/articles/tools/black-box/261-BSI.html>.
- <http://opensourceTesting.org>: *Open source testing tools, news and discussion*.
- <http://msdn.microsoft.com/security/securecode/threatmodeling/acetm/>.
- Ghosh, Anup K.; O'Connor, Tom; & McGraw, Gary: *An Automated Approach for Identifying Potential Vulnerabilities in Software*.
- Oakland, California, May 3-6, 1998. Los Alamitos, CA: IEEE Computer Society Press, 1998: *Security and Privacy*.
- TechRepublic Publication: *Risk Based Security Testing*.
- Grance, T.; Myers, M.; & Stevens, M: *Security Considerations in the Information System Development Life Cycle*, (NIST Special Publication 800-64), 2004.
- December 2007, HP: *Pillars of Application Quality: Security, Functional, and Performance Testing*.
- Peter Varhol, Technology Strategy Research, LLC: *Building Secure Web Applications*.
- Bruce potter and Gary McGraw: *Software Security Testing*.
- C. C. Michael and Will Radosevich: *Risk-Based and Functional Security Testing*.
- Guttman, Barbara; Roback, Edward: *An Introduction to Computer Security*.
- Gearhead Product Security Trainings for Whitebox Testing.
- Wenliang Du & Aditya P. Mathur: *Testing for Software Vulnerability Using Environment Perturbation*. Center for Education and Research in Information Assurance and Security (CERIAS), CERIAS Center and Software Engineering Research Center (SERC).
- Du, W. & Mathur, A. P.: *Vulnerability Testing of Software System Using Fault Injection* (COAST technical report). West Lafayette, IN: Purdue University, 1998.
- Dustin, Elfriede; Rashka; Jeff; McDiarmid, Douglas; & Nielson, Jakob: *Quality Web Systems: Performance, Security, and Usability*.
- Wack, J.; Tracey, M.; & Souppaya, M.: *Guideline on Network Security Testing* (NIST Special Publication 800-42), 2003.
- NIST. *The Economic Impacts of Inadequate Infrastructure for Software Testing* (Planning Report 02-3). Gaithersburg, MD: National Institute of Standards and Technology, 2002.
- Howard, Michael & LeBlanc, David: *Writing Secure Code*, 2nd ed. Redmond, WA: Microsoft Press, 2002.
- Hoglund, Greg & McGraw, Gary. *Exploiting Software: How to Break Code*. Boston, MA: Addison-Wesley, 2004.
- Grance, T.; Myers, M.; & Stevens, M. *Security Considerations in the Information System Development Life Cycle* (NIST Special Publication 800-64), 2004.
- Graff, Mark G. & Van Wyk, Kenneth R. *Secure Coding: Principles and Practices*. Sebastopol, CA: O'Reilly, 2003 (ISBN: 0596002424).
- Viega, John & Messier, Matt. *Secure Programming Cookbook for C and C++*. Sebastopol, CA: O'Reilly, 2003 (ISBN: 0596003943).
- G. Hoglund and G. McGraw, *Exploiting Software*, Addison-Wesley, 2004.

Why Tests Don't Pass

Douglas Hoffman
Doug.Hoffman@acm.org

Abstract

Most testers think of tests passing or failing. Either they found a bug or they didn't. Unfortunately, experience shows us repeatedly that passing a test doesn't really mean there is no bug. It is quite possible for a test to surface an error but it not be detected at the time. It is also possible for bugs to exist in the feature being tested in spite of the test of that capability. Passing really only means that we didn't notice anything interesting.

Likewise, failing a test is no guarantee that a bug is present. There could be a bug in the test itself, a configuration problem, corrupted data, or a host of other explainable reasons that do not mean that there is anything wrong with the software being tested. Failing really only means that something that was noticed warrants further investigation.

The paper explains the ideas further, explores some of the implications, and suggests some ways to benefit from this new way of thinking about test outcomes. The paper concludes with examination of how to use this viewpoint to better prepare tests and report results.

Biography

Douglas Hoffman has over 30 years experience as a consultant, manager, and engineer in the computer and software industries based on a solid foundation in computer science and electrical engineering. He provides organizational assessments, strategic quality planning, and test planning services. His recent technical work has focused on test oracles and advanced automation architectures. He is an ASQ Fellow, member of ACM and IEEE, holds ASQ Certificates in Software Quality Engineering and Manager of Quality/Organization Excellence, and has been a registered ISO Lead Auditor. He holds credentials for teaching Computer Science at the college level and has done so at the University of San Francisco, UC Santa Cruz Extension, and Howard University.

Douglas is a Past Chair of the American Society for Quality (ASQ) Silicon Valley Section and the Santa Clara Valley Software Quality Association (SSQA), a task group of the ASQ. He is a member of the Board of Directors in the Association for Software Testing (AST) and committee member for the Pacific Northwest Software Quality Conference (PNSQC). He is also a regular speaker at software quality conferences including STPCon, STAR, PNSQC, Quality Week, and others.

Copyright © 2009 Software Quality Methods, LLC.

Published at the Pacific Northwest Software Quality Conference 2009

Why Tests Don't Pass¹

Douglas Hoffman

Summary

When we execute a test the results come down to either finding a bug (the software failed the test) or not finding a bug (the software passed the test). We either have something to investigate/log into the bug database or not. Unfortunately, experience repeatedly shows us that passing a test doesn't really mean that there are no bugs present. It is possible for the test to miss the bug. It is also quite possible not to notice an error even though a test surfaces it. Passing a test really means that we didn't detect anything out of the ordinary.

Likewise, failing a test is no guarantee of the presence of bugs. There could be a bug in the test itself, a configuration problem, corrupted data, or a host of other explainable reasons that are not due to anything wrong with the software being tested. Failing really only means that something that was noticed warrants further investigation and possible reporting.

This paper explores these ideas and some of the implications further and suggests some ways to benefit from this new way of thinking about passing and failing. The talk also examines how to use this viewpoint to better prepare tests and report results.

Why Tests Don't Pass (or Fail)

Some working definitions will help clarify what I mean in this paper:

A **Test** is an exercise designed to detect bugs. It is generally composed of set-up, execute, clean-up, and initial analysis of results.

A test **Passes** when the software works as expected. Here, nothing was noted during test execution or initial analysis.

A test **Fails** when it indicates there is an error in the software under test (SUT). Here, something was noted either during test execution or in the initial analysis of the outcomes.

A test **Oracle** is the principle or mechanism by which we decide whether the SUT behavior is good or bad.

We make observations about the general behavior of the system and specific behaviors related to the exercise when we run a test. These observations are evaluated through one or more oracles. General behavior may include such things as memory leaks, display behavior, network access, etc. that are not specifically being tested but could manifest errors from bugs present in the SUT. The specific behavior is directly related to the test exercise and the expected software behaviors from the exercise. The *results* are the elements explicitly checked as part of the initial test analysis.

¹ Extending work first presented at the Conference for the Association for Software Testing (CAST) 2009

Test Result Possibilities

There are two outcome alternatives when we run a test: we observe behaviors that, according to our oracles, indicate (or don't indicate) that the software is buggy. Further action is indicated when the test results flag us to an error or we observe something unexpected during the running of the test. After the initial failure indication, further investigation of the unusual condition is necessary to determine whether or not there is an underlying bug. When the test is complete and we've done the initial analysis, we know whether we will pursue any bugs. In 20/20 hindsight we may know whether or not a bug exists.

A test passes when there are no indications of any problem. In this case, there is nothing to do except record that the test was run.

Either way, though, there may or may not be an error in the SUT. After we have a failure, investigation may show a reportable bug. The investigation may also show that an error indication is due to one of many possible other causes. We don't automatically report all failing tests specifically because of these false alarms. (False alarms and some of their sources are explored further below.)

Table 1 illustrates the two possible outcomes and four possible states after running a test. There are two possible situations regarding bugs in the SUT: either there are no bugs in what we are testing, or there are bugs an excellent test should expose. When we run a test, our initial analysis indicates either pass or fail. There are four combinations of the two variables. When a test passes we are unaware of any need for further investigation, regardless of whether or not a bug is actually present. Doing nothing more is the correct action if there are no errors present. However, we will also take no action if there are errors present but unnoticed (a Type I error). This case I call a "Silent Miss" because we do not know about any error from this test and therefore will quietly move on.

Real Situation		
Test Result	No Bug in SUT	Bug in SUT
As Expected (Pass)	Correct Pass	Silent Miss
Abnormal (Fail)	False Alarm	Caught It!

Table 1

A failure tells us that further investigation is required. There are the same two possible situations regarding bugs in the SUT: either there are no bugs in what we are testing, or there are. We investigate when a failure occurs, so we will find out which is the case. It's a good thing if we find a bug. That's the purpose of having the test and it has correctly identified that a bug is present. A "False Alarm" occurs when a test indicates a failure but there is no bug in the SUT (a Type II error). In this case we investigate because of the test failure, and through the investigation we discover that there is nothing apparently wrong with the SUT. Reasons for the failure could be an error in the test or some condition that caused unexpected but normal behavior from the SUT. Unexpected because something happened that was different from the anticipated behavior of a bug-free SUT (bug free relative to what the test is looking for). Normal because, by knowing the cause we see that the results are what we expect should happen. Having a false alarm is not good. The time spent running the test, analyzing the results, and investigating the suspected failure tells us nearly nothing about the quality of the SUT. The time invested results in almost no new information about the quality of the SUT.

A Model for Software Test Execution

It is important for a tester to realize that the four situations exist. There are many reasons for silent misses and false alarms. The model of test execution below helps explain why we can have undetected errors in the face of good tests and why some errors are intermittent or not reproducible.

The software test execution model includes the elements that impact SUT behavior (such as data and system environment) and helps articulate what to look at to verify test results. (See Figure 1.) Understanding and identifying the elements leads to more creative test design and better validation of outcomes.

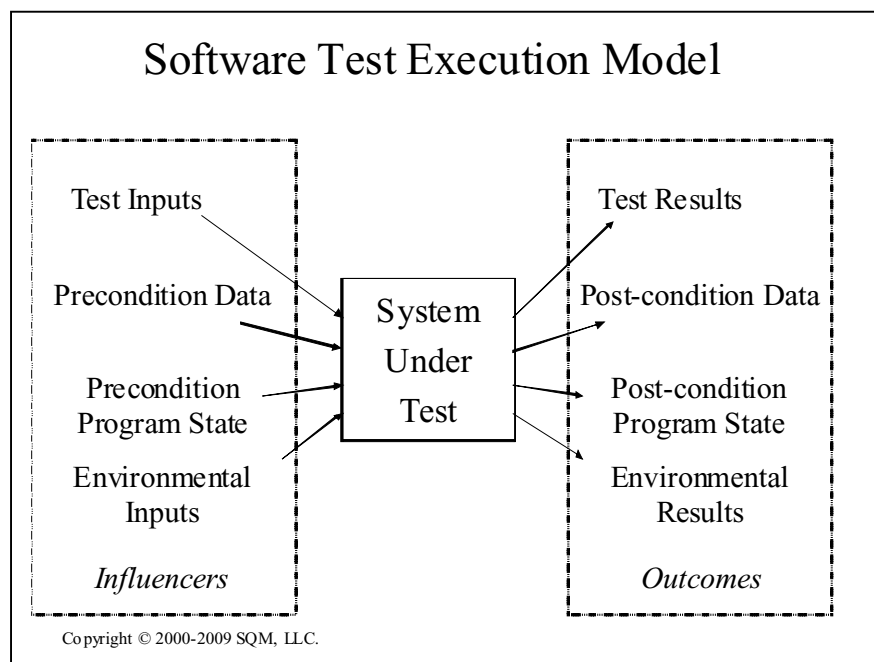


Figure 1

The four areas that influence the SUT behavior (Influencers) are test inputs, precondition data, precondition program state, and the system environment. There are corresponding areas where the SUT may have outcomes that should be verified (Outcomes). The areas are described briefly in Table 2.

Terms	Definition
Test Inputs/Results	The test exercise and expected results from proper behavior of the SUT. (This is what most testers think of as the test.)
Precondition/Post-condition Data	Files, databases, and interim values used or set by the SUT. (This includes <i>any</i> data that <i>could</i> be accessed by the SUT when a defect is present.)
Precondition/Post-condition Program State	The internal program state variables. Two examples are an error dialog and an input screen. (Many of the program state variables are internal to the SUT and cannot be easily set or detected.)
Environment	System factors such as hardware components, memory size, network characteristics, operating system, other running programs, user permissions, etc.
Influencers	Elements that affect the SUT behavior. This includes the test inputs, precondition data, precondition program state, and the environment. (Note that influencers include all elements that possibly affect the behavior of the SUT when bugs are present, and therefore the scope is infinite.)
Outcomes	Elements that are affected by the SUT behavior. This includes the test results, post-condition data, post-condition program state, and the environment. (Note that outcomes include all elements possibly affected by the behavior of the SUT when bugs are present, and therefore the scope is infinite.)

Table 2

The influencers include an infinite number of possibilities in the three areas we don't generally control: precondition data, precondition program state, and environment. What we control are the test inputs. (Often we validate or control some of the input data, program state, and environment, but this is limited to specific elements required by the test.) We cannot control all the factors in any of the three domains, especially in light of possible SUT errors that may reference influencers we never considered (e.g., opening the wrong file). When we cannot duplicate an error we did not monitor or control a critical influencer.

Likewise, there is an infinite space of possible outcomes in the data, program state, and environment we don't normally consider, decide not to check, or miss in our analysis and test design. Again, this is especially true in light of possible SUT errors that may affect outcomes the SUT's intended behavior would not touch (e.g., writing to the wrong file). We will not detect an error if we don't check the outcomes where the error manifests itself.

Questions to Ask Ourselves

There are a number of questions we can ask that will improve our tests based upon the software test execution model. Answering these questions may improve the quality of tests.

- ❑ What values/conditions should influence the SUT? How do we set/cause them? (What influencers are we not controlling or monitoring?)
- ❑ Are we controlling or monitoring the best influencers?
- ❑ Are we checking the most important outcomes? (What outcomes do we know about that we're not checking?)
- ❑ How do we know the expected outcomes?
- ❑ What gives us confidence that the test isn't missing real bugs?
- ❑ What values/conditions should influence the SUT? How do we set/cause them? (What influencers are we not controlling or monitoring?)
- ❑ Are we controlling or monitoring the best influencers?

The behavior of the SUT is controlled by a multitude of outside influences, many of which we take for granted. To the degree possible, we should be aware of the potential for external influences on the SUT, both those required for the test to run and those outside of the ones specifically required by the test. This becomes particularly relevant for automated tests, but applies to manual tests as well. We need to decide whether to control them, monitor them, or ignore them for the purposes of running the test exercise.

The question also comes up as to how we can monitor or control them if we want to. System resources may become unavailable after the start of testing, data values changed, etc. These things may be outside of our control. Monitoring the values may be possible, but expensive. But, even if we decide not to monitor or control, we are much more capable of investigating and isolating bugs if we are aware of the potential for external factors to influence SUT behavior.

Are we checking the most important outcomes? (What outcomes do we know we are not checking?)

How do we know the expected outcomes?

Likewise, the possible outcomes from running the test are infinite in the face of potential SUT bugs and influencers. Being aware of potential outcomes, however, does not address the problem of having test oracles to predict or analyze them. Thinking about the potential outcomes gives us a much better chance to monitor and verify them and thus detect more potential SUT bugs. Being aware of possible outcomes also provides us with more information when investigating and fault isolating when unexpected behavior is detected.

What gives us confidence that the test isn't missing real bugs?

Lastly, we should ask ourselves if we have done everything practical in the test to maximize the chance that the exercise might surface and detect a bug if it's present. Besides answering the question of whether or not this is the best exercise, we need to evaluate what we should know, record, and control about influencers and what outcomes we should evaluate.

Implications

There is a possibility for bugs whether a test indicates passing or a failing. A silent miss occurs when we aren't checking the affected outcomes, the test has a bug, or we don't notice a failure the test exposed. The bug may be detected later through some other process, but the test results indicated that no further action is required. This means that we should be skeptical when a test indicates a "pass." It's always possible in spite of a pass indication that there is still a bug in the SUT in the area under test.

There is a possibility that there are no bugs even though a test fails. A false alarm could be due to many causes; one or more of the influencers cause unanticipated but normal outcomes (for the SUT, given the values or events of the influencers). At the end of our investigation we conclude that there is nothing wrong with the SUT. This means that we should be skeptical when a test indicates a "fail." It is always possible that there is no bug in the SUT in spite of a failure indication.

Pass/Fail metrics don't really give us interesting information until the cause of every pass and fail is understood. Unless we validate the pass indications, we won't know which tests missed a bug in the SUT. But, we have no reason to do so. Because we don't really know which passes are real (and are unlikely to investigate to figure out which are), any measure of the count of passes misrepresents the state of the SUT with regard to the testing. Likewise, the count of failures is only meaningful after thorough investigation and identification of which failures are due to SUT errors and which not.

Skepticism is healthy when reporting test results. As much as we'd like to have definitive answers and absolutes, the results from our testing are inconclusive, especially before failures have been completely investigated. Initial test reports should be qualified with the fact that the numbers are subject to change as more information is gathered about failing tests. Just as the government regularly revises past economic indicators when new information is available, so should we treat passes and failures as estimates until we have gathered all the evidence we expect to glean from the tests.

Conclusions

There are lots of reasons for "passing" or "failing" a test other than bugs in the SUT. Simply stated, "pass" doesn't mean the SUT passes and "fail" doesn't mean the SUT fails. They are indications of whether or not further investigation is necessary. Tests that pass provide no reason to investigate; the results were consistent with the SUT behaving as expected. There may still be problems in the area of the SUT that test focuses on. Whether the problems were surfaced by the exercise or not, pass means we didn't note anything out of the ordinary, not that the SUT works.

Failing tests provide reasons to investigate the SUT because the test turned up something unexpected (or an expected indicator of a bug). Through the investigation, we will decide whether there is a bug present or if this is a false alarm. Finding a bug is a good thing - that's the purpose of testing, after all. In the case of a false alarm, we have learned nothing new about the SUT from the test. The problems surfaced by a fail don't always come from SUT errors, so failing a test doesn't mean that there is a bug present.

We aren't checking all of the results, so we know we may miss some errors. We don't know the outcomes from arbitrary errors, so we can't know all the right results to check.

Pass/Fail metrics don't really give us interesting information initially. "Passes" include an unknown number of actual SUT bugs. "Fails" overstate the number of SUT bugs until all the failures are isolated and causes identified.

By asking questions about the influencers and outcomes we can create better tests and do better testing. Knowing more about them can lead to better reproducibility, catching more bugs, and improved fault isolation.

Visualizing Software Quality

Marlena Compton
Equifax, Inc.
marlena.compton@equifax.com

Marlena Compton automates tests and performs testing in a distributed systems group at Equifax where she has worked for the past 5 years. She will be receiving her Software Engineering MS in December of 2009. This paper is derived from her thesis on visualizing software quality. She blogs about technical testing and data visualization at marlenacompton.com

Abstract

Moving software quality forward will require better methods of assessing quality quickly for large software systems. Assessing how much to test a software application is consistently a challenge for software testers especially when requirements are less than clear and deadlines are constrained. In assessing the quality of large-scale software systems, data visualization can be employed as an aid. Treemap visualizations can show complexity in a system, where tests are passing vs. failing and which areas of a system contain the most frequent and severe defects. This paper describes the principles of data visualization, how treemap visualizations use these principles and how treemaps can be employed for making decisions about software quality.

1. Introduction

For most software testers obtaining data is no longer a problematic task. Between test case management and bug tracking software, most testers can easily produce lists of highly critical bugs created from running mountains of test cases multiple times. Software testers have never before had so much access to so much data. But what does this data say about the health of a system?

Wading through bug reports and tests to produce a meaningful answer about the SUT's (System Under Test) overall health or health in specific areas can take more time than a project plan allows. Eventually, the task becomes boiling down all of the data into communication that is concise enough and clear enough to be understood in the limited review time available, yet still illustrates the problem with accuracy. As a means of communicating large sets of data with integrity, data visualization can help testers communicate their diagnosis of a system in a concise manner.

Let's begin with an historical example, albeit from a field far from software testing. During August of 1854, Dr. John Snow of London found himself in the center of just such a situation. Faced with a public fleeing the city's central district in sheer panic over a quickly spreading outbreak of cholera, Snow saw in the epidemic an opportunity. Despite the prevailing theory of the time that cholera was spread through contact with corpses or by breathing putrefied air, Snow suspected (and was determined to prove to a skeptical medical community) that the disease was spreading via a contaminated water source [1].

Beyond the curtailing of a cholera outbreak, there was an even more important reason for Snow to convince London authorities that cholera was spread through water. A few years earlier, in 1848, a law had been passed requiring that all waste in the city of London was to be deposited into a sewage system that flushed the waste directly into the River Thames which was also the source of drinking water for Londoners. Snow was convinced that London was on the verge of an even deadlier health epidemic. He used data collected from the 1854 epidemic and data visualization to prove this.

To effectively employ data visualization in making and communicating decisions about system quality, it is important to have an understanding of the process involved and the qualities in a good visualization that maintain the integrity of the data being presented. In this paper, we will begin with the above historical case study that demonstrates how a good map can communicate a complicated data set. We will then trace the path from understanding what makes a good visualization and the process involved in creating a good visualization to how a quality professional can apply these techniques for visualizing software quality, including tests and defects, in a system.

In section 2, we will summarize the characteristics of excellent visualizations with integrity, based on the work of one of the foremost data visualization experts, Prof. Edward Tufte. In section 3 an analogy will be drawn between John Snow's map of a cholera epidemic and maps that can help testers communicate system problems. In section 4, Treemap visualizations will be introduced. The way treemaps embody Tufte's principles of data visualization and how they are used to visualize source code metrics will be explained. Section 5 will show how treemaps can be applied to quality data such as tests and defects. Finally we will conclude with a discussion of what lies ahead for the visualization of software quality.

2. Data Visualization Principles

Before data visualization can be employed for *any* purpose, there must be an understanding of what constitutes a good visualization. In his pioneering book, The Visual Display of Quantitative Information [2], Edward Tufte explains the underlying principles of good data visualizations. At the highest level, visualizations should reveal *data*. High quality visualizations represent extremely large sets of numbers in a very small space. The data they represent is clear to the viewer and free of distortion. A good visualization allows the viewer to easily compare the information it is presenting. Data can be studied with the finest micro level detail, but also forms another layer of information at the macro level. Everything in an excellent visualization fits together like pieces of a puzzle in much the same way as a great painting or other work of art.

Tufte introduced several terms used in describing visualizations. **Data ink** refers to ink in a graphic that represents the statistics or other information in the graphic. It excludes marks such as gridlines. The **data ink ratio** is the amount of data ink divided by total ink used to print a graphic. The higher the data ink ratio, the greater the area of a graphic that is used to depict data. **Chartjunk** is extra decoration that appears in a graphic, but is not related to the information contained in the graphic.

Integrity in data visualization is very important. There are several principles that apply when assessing the integrity of a graphic. Labeling should be used extensively to dispel any ambiguities in the data. Explanations should be included for the data. Events happening within the data should be labeled. All graphics must contain a context for the data they represent.

Numbers represented graphically should be proportional in size to the numeric quantities they represent. The number of dimensions carrying information (size, color, location) should not exceed dimensions in the data. Variation should be shown for data only and not merely for the design of the graphic [2].

Section 3. Software Testing and the Visualization of an Epidemic

Communicating illness in a system to a business team eager for profits is a difficult task. All project managers have schedules, and if software quality threatens to derail the schedule, denial is often a popular fallback strategy. Thus, any visualization of quality must show problems in the system so explicitly as to be undeniable.

John Snow's map of a cholera outbreak accomplished this task and ultimately benefited the city of London with a much cleaner supply of drinking water. Figure 1 is taken from a reproduction of his map. Victims are marked with stacks of black lines (think black coffins piled high). The affected water pump is marked with a dot. Notice that there are no black marks at the neighborhood brewery, situated in the upper-right corner.

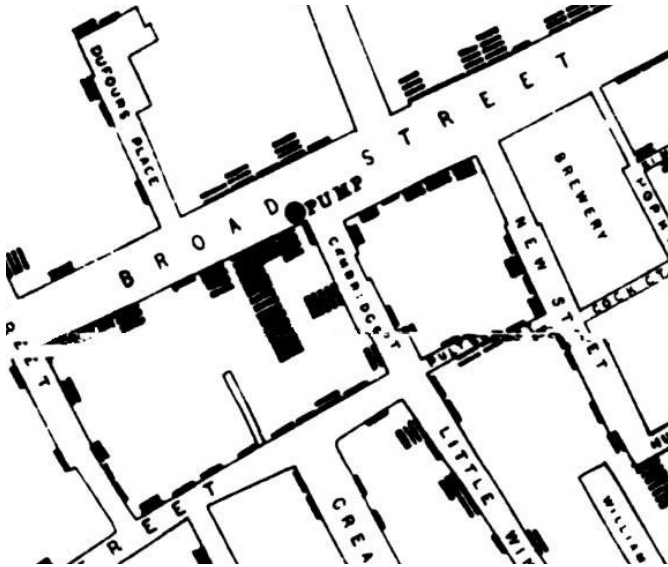


Figure 1. John Snow's Cholera Map

There are no marks at the brewery because the workers on-site were drinking their allotment of beer instead of drinking water from the contaminated Broad Street pump.

Snow's map is a bellwether not only because it proves, undeniably, a theory that the General Board of Health in London actively resisted, but because it is an example of compiling large amounts of data into a visualization depicting a dire problem. Snow had a list detailing the names and addresses of 83 people who had died from cholera. He also had an invaluable resource of detailed information on the neighborhood's residents in the form of local clergyman Henry Whitehead. While Whitehead tracked down and questioned everyone he possibly could about their drinking habits, Snow analyzed this data to form patterns of who had been *drinking* the water, who had *died* and, just as importantly, who had *not died*. Reverend Whitehead's authority and John Snow's data provided enough evidence for the parish board to remove the pump handle from Broad Street, which likely contributed to the end of the 1854 epidemic. Yet London's wider health community, along with the General Board of Health, remained unconvinced. Rather than attempting to share each and every story, Snow reduced the information into the map shown in Figure 1. If this type of visualization technique could improve the quality of a city's water, it can also be beneficial for the quality of software.

Moving forward approximately 150 years to the beginning of the 21st century, testers swim in a rising tide of tests and a sea of automated builds provided for us by the software testing tools of our highly connected, scripted and automated world. The number of tests for a system swells quickly. The amount of time between builds and releases continuously shrinks. In John Snow's time, the science of Epidemiology (the study of epidemics) was just beginning. The individual patient was the micro view of health whereas the health of a city or town was becoming the macro view of health. This required having access to data, such as a list of people who had died from cholera and where they had obtained their drinking water. The same sea change is now occurring in testing. One occurrence of a passing test is now a micro-level data point in a lengthy list of tests for components and sub-components executed several times. We now need a macro-level view of our tests. We need to see problems and complexities for our code, tests and bugs as they relate to a SUT's geography. Testers are in need of a *map*.

4. Using Treemaps to Visualize Source Code

4.1 Treemaps

The data measures, or graphic elements, on a map can perform multiple functions by showing both content and form. In the case of geography, for example, they show the shape and location of geographic units, whether these are a stream, road or topographical change in height. In an abstract sense, color and shading on a map can also indicate the level of a variable. Maps can thus contain very dense amounts of information in extremely small spaces. They can be examined at the macro and micro levels, revealing large, overall patterns across countries, states or lines of code. On the detail contained in maps, Edward Tufte writes, "The most extensive data maps... place millions of bits of information on a single page before our eyes. No other method for the display of statistical information is so powerful" [2].

A treemap is a visualization with an extremely high data-ink ratio used to represent hierarchically structured data. The viewer sees the entire structure of a tree on one screen, no matter how many branches and leaves occur within the tree's structure. Chartjunk is minimized. The first treemap display was created by Prof. Ben Shneiderman for displaying files on his computer [3].

Treemaps consist of nested rectangles representing the branches and leaves in a hierarchical structure. Branches form the borders and nesting within a treemap. The size and color of the boxes is based on attributes of the leaves. The size of any rectangle depends on some type of weighted factor. In the case of a treemap representing a file system, the directories and sub-directories form the borders, the sizes of the rectangles are based on file size and the color shows the file type. Larger directories take up more space in the visualization[3].

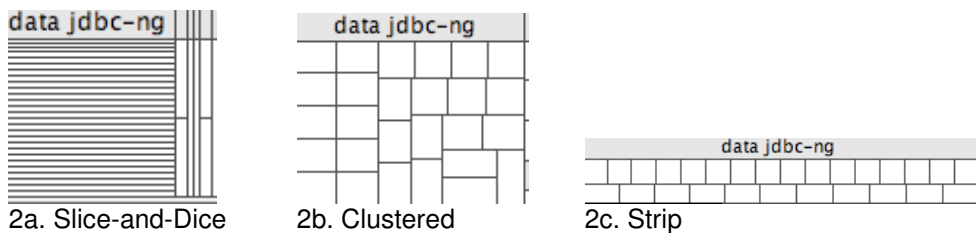


Figure 2. Treemap Layouts

The algorithm used to determine how the rectangles will be laid out on the screen is called the layout algorithm. The three most widely used and studied types of layouts are the original "slice-and-dice" algorithm and two variants, the "clustered" algorithm (sometimes referred to as "squarified") and the "strip." The slice-and-dice layout uses parallel lines to divide a rectangle representing a larger object into smaller rectangles representing sub-objects. At each level, the orientation of the lines are switched from vertical to horizontal. The clustered layout uses recursion to produce rectangles that have more of a uniform square shape than the slice-and-dice layout. The strip layout produces rectangles that are in order like the slice-and-dice layout but have a better aspect ratio for viewing like the clustered layout. Horizontal rows of rectangles are created in the strip layout [4].

Each layout has advantages and disadvantages. The slice-and-dice layout preserves the order of the data being represented and so produces treemaps in a consistent format. However, it creates long, skinny rectangles that are difficult to see, select, label and compare. While clustered layout treemaps are easier to view, if data is added to the dataset, order is not preserved and the entire layout changes. Maintaining the order within a dataset can be a key aspect when a viewer is looking for patterns in the data [4].

4.2 Metrics and Software Visualization

Bugs are often found by testing areas of code or features that are more complex. In using software metrics to assess code risk, also known as "code smell," testers often employ software metrics such as loc (lines of code) count, cyclomatic complexity, and CK (Chidamber and Kemerer) Metrics. Johnston et al point out that these metrics can sometimes be misleading or that they don't "tell the whole story. " Comparing metrics, however, can give testers a more complete picture of where the "smells" are [6].

Treemaps can visualize a comparison of metrics by using the size of a rectangle for one metric and the color of a rectangle for a second metric. For example, figure 3 shows a treemap visualizing java source code. The geography of the system is preserved because the hierarchy of the packages and classes is maintained. The size of the rectangles is determined by the loc count for each class and the color is determined by the weighted-methods per class (wmc).

WMC (Weighted Methods Per Class) is a metric proposed by Chidamber and Kemerer in their classic paper, "A Metrics Suite for Object Oriented Design." To calculate this metric, a complexity (specifically left by Chidamber and Kemerer to the reader) is applied to each method in a class. These complexities are then summed as weighted methods per class. If no complexity is applied to the methods, the metric simply becomes the number of methods [5].

By visualizing a comparison of software metrics for a system under test (SUT), testers can make better decisions about areas of code more likely to contain bugs or that would be more complicated to test. Figure 3 shows the component "build" in a SUT. Comparisons can be made about the sub-components. The largest class with a higher WMC complexity is JobModuleConfigurePanel. This class is larger and more complex than JobXMLContainer. JobXMLContainer is larger than APMModuleConfig, but it has a lower WMC complexity.

build					
JobModuleConfigurePanel	JobModuleSelectPanel	JobInfoBuildPanel	JobBasicInfoPanel	JobBuildForm	
	DPAAttributeModelPanel	JobBasicInfoPanel	APModuleConfig	JobOutputConfigPanel	APModuleConfigPanel
JobModuleTreePanel	JobXMLContainer	APModuleConfigPanel	DPMModuleConfigPanel	APModuleConfigPanel	JobModuleSelectPanel
				XMLContentS	JobModulePanel

Figure 3 Java Source Code and CK Metrics

Figure 3 is an illustration of areas where tradeoffs occur when assessing the need for testing resources. A component that is larger or more complex will require more test cases, more time and possibly more testers. If more resources are used for testing the larger components, this means that less resources will be available to test smaller components. Figure 3 shows that some of the smaller components in this SUT, despite their size, have a higher complexity than the larger components, and therefore, a higher risk of containing bugs. Minimizing the testing of JobBasicInfoPanel, APModuleConfig and JobOutputConfigPanel is less desirable once complexity is taken into account. It is possible to boil both the complexity and size down to one number, but this would create an oversimplification for a complex situation. Seeing loc and complexity together allows for more informed decisions about the amount of testing required.

5. Visualizing Quality with Treemaps

5.1 Visualizing Tests

Producing a treemap based on source code and source code metrics is fairly easy. Given the availability of data from a test case management system, it is also possible to create treemaps of tests and treemaps of defects. This begs the question, what would one want to discover about tests that would best be suited for the format of a treemap?

Mozilla uses the test case management software, Litmus, to track all of their testing in an online environment. This allows for volunteer participation and means that the vast majority of their test cases and test runs are available for viewing online. Currently, test cases are primarily shown 1 web page at a time. There are a few reports that show more tests, including most recent failures which shows failing test cases in groups of 50 at a time.

Figure 4 is an excerpt from a treemap visualizing 3859 Mozilla tests. This treemap shows the status of most of Mozilla's tests. The size of each rectangle is determined by the number of test steps in a test case. The color is determined by the number of failing test runs.

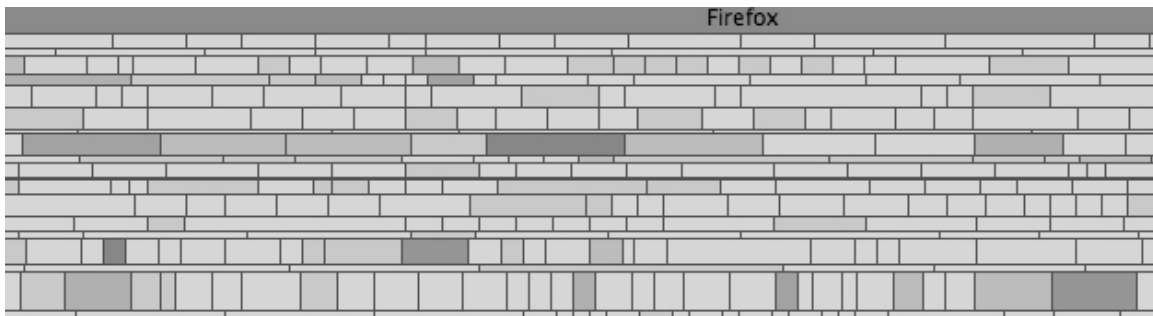


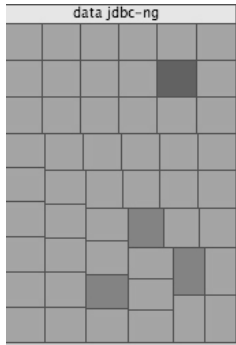
Figure 4. Approximately 150 of 3859 Firefox Tests in a Strip Treemap

Figure 4 uses the strip layout because this layout preserves the order of the data. This is in contrast to the clustered layout used to visualize source code. The clustered layout produces more uniform squares, but at the compromised loss of any order in the data represented. For viewing failed tests, preserving the order of the data has the benefit of showing clusters of failures. This provides a macro/micro view of where tests are failing at the higher product level or at the more finely grained level of individual tests.

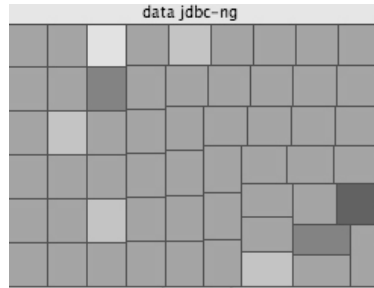
Figure 4 has a couple of issues. Data used in this treemap was parsed from html. Some tests had to be discarded because of badly formed html. This could have an effect on the accuracy of the failure clusters by de-emphasizing areas with failures that are not included. While the number of test steps can give a vague indication of test size, it is a very weak indication. This does not address how long a test may take to run or the complexity of the steps needed to run the steps. Plenty of research exists for source code complexity, but more research is needed for test case complexity.

5.2 Visualizing Defects

Defects visualized in a treemap can show areas of software that contain the most severe bugs and the most volume of bugs. When organized according to bug status, a treemap of bugs can show a bug's status in the bug lifecycle. If open and closed bugs are shown side by side, it is possible to see which areas of a system under test have had very severe bugs in the past and compare that with the severity of current bugs in the same area.



5a. Open Defects



5b. Closed Defects

Figures 5a and 5b show sections of a treemap created from the open source project, GeoTools. Figure 5a shows open defects for the component data jdbc-ng. Figure 5b shows closed defects for the same component. All of the defects are equally sized, and color is used to show severity. As the severity of a bug increases the color showing the severity becomes darker. Although 1 high severity defect and 2 medium severity defects have been closed, there is 1 severe defect remaining.

6. Conclusion and Future Directions

In this paper, I have introduced the concept of data visualization and how it can be useful for moving software quality forward. I have summarized Tufte's principles for data visualizing large data sets of multi-variate data. I have examined how the quality of source code is already visualized using CK metrics and applying Edward Tufte's principles of data visualization. Following that, I showed how this could also be applied to software quality data to help testers make decisions.

There are many opportunities for the future use of treemaps in testing. Some work has been to show antipatterns in treemaps [7], but this work is minimal. Future opportunities for visualizing quality branch out much further than treemaps. Performance testing was not included in this discussion but is ripe with opportunities particularly in the use of a stacked graph or a horizon graph. Stacked graphs have most recently been used to visualize music listening history on last.fm [8]. The horizon graph addresses a weakness of the treemap because it shows the time series for large numbers of components by using Edward Tufte's concept of small multiples [9].

With some involvement from the field of artificial intelligence, not only could information such as test status and hierarchy be visualized, but the semantic content of tests could also be analyzed. The semantic data of tests and bugs could be mined and filtered to produce relationships among tests in structures such as a network graph.

Even though we are on the verge of using these newer types of visualizations, the basic principles of visualization should not be forgotten. They should be kept in mind and ready for use, even for charts as seemingly mundane as a test execution report. By practicing good visualization techniques with simpler charts and graphs we will have the ability "at the ready" to make the creative leap necessary for more complex forms of visualization that are rapidly becoming more necessary.

7. References

- [1] Johnson, Steven. *The Ghost Map: The Story of London's Most Terrifying Epidemic - and How it Changed Science, Cities and the Modern World*. New York: Riverhead Books, 2006.
- [2] Tufte, Edward R. *The Visual Display of Quantitative Information*, 2nd ed. Cheshire, Connecticut: Graphics Press, 2001.
- [3] Johnson, Brian and Ben Shneiderman. "Treemaps: A Space Filling Approach to the Visualization of Hierarchical Information Structures." *IEEE Conference on Visualization, San Diego, California, Oct. 22-25, 1991*. Ed. Gregory M. Nielson, Larry Rosenblum. Los Alamitos, CA: IEEE Press, 1991.
- [4] Bederson, Benjamin B., Ben Shneiderman and Martin Wattenberg. "Ordered and Quantum Treemaps: Making Effective Use of 2D Space to Display Hierarchies." *ACM Transactions on Graphics*. 21.4 (2002): 833-854.
- [5] Chidamber, Shyam R. and Chris F. Kemerer. "A Metrics Suite for Object Oriented Design." *IEEE Transactions on Software Engineering*. 20.6 (1994): 476 - 493.
- [6] Johnston, Ken, Alan Page and B.J. Rollison. *How We Test Software at Microsoft*. Redmond, Wa: Microsoft Press, 2009.
- [7] Langelier, Guillaume, Houari Sahraoui and Pierre Poulin. "Visualization-based Analysis of Quality for Large-scale Software Systems." *IEEE/ACM international Conference on Automated Software Engineering, Long Beach, Ca, Nov. 7-11, 2005*. Ed. David Redmiles. New York: ACM Press, 2005. Pages: 214-223.
- [8] Byron, Lee and Martin Wattenberg. "Stacked Graphs – Geometry & Aesthetics." *IEEE Transactions on Visualization and Computer Graphics*. 14.6 (2008): 1245-1252.
- [9] Agrawala, Maneesh, Jeffrey Heer and Nicholas Kong. "Sizing the Horizon: The Effects of Chart Size and Layering on the Graphical Perception of Time Series Visualizations." *International Conference on Human Factors in Computing Systems, Boston, Ma, April 4-9, 2009*. Ed. Dan R. Olsen, Jr., Richard B. Arthur. New York: ACM Press, 2009. Pages 1303-1312.

Everyone Can Test Performance

Scott Barber

sbarber@perftestplus.com

Abstract

Many people believe that expensive tools and complicated techniques are required to test performance. My advice to “start by becoming a mid-level everything,” frequently offered to those looking to break into performance testing, is consistent with that belief. Cem Kaner’s quip of “You’re going to fill 150 pages with the phrase ‘Hire a consultant’ in a bunch of different languages?” when I mentioned that I was considering writing a how-to style book on performance testing, also seems to support that belief.

The fact is that there is at least some truth behind the common belief that expensive tools, complicated techniques, performance testers who are at least mid-level “everything” and likely an expert consultant are all needed to accomplish relatively complete performance testing on high-risk, high-profile, high-usage applications built on leading edge technology platforms. That, however, does not negate the reality that a lot of valuable performance related information can be uncovered with exactly the tools and knowledge you have at your disposal right now. In my experience, most of the performance related information needed by stakeholders to make sound decisions and that development teams need to dramatically improve system performance is easily obtainable by performance testing laypersons.

I view performance testing as an empirical technical investigation conducted to provide stakeholders with information about the quality of the product or service under test with regard to speed, scalability and/or stability characteristics. Much of the information needed to achieve acceptable speed can be collected by the performance testing layperson. The information needed to achieve acceptable scalability and stability is where the tools, techniques, knowledge, and consultants are typically needed, but even in these areas the performance testing layperson can make significant contributions.

In most cases, however, teams and organizations become so focused on the complicated parts of performance testing that they completely overlook the value of the “easy stuff.” This presentation explores several techniques that the performance testing layperson can put to use to speed up and simplify the collection of valuable performance related information about your software.

Biography

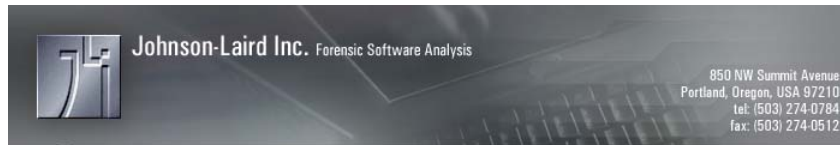
Scott Barber is the Chief Technologist of PerfTestPlus, Executive Director of the Association for Software Testing Co-Founder of the Workshop on Performance and Reliability and co-author of the Microsoft patterns & practices book Performance Testing Guidance for Web Applications. Scott thinks of himself as a tester and trainer of testers who has a passion for testing software system performance.

In addition to performance testing, he is particularly well versed in developing customized testing methodologies, embedded systems testing, testing biometric identification and personal security systems, group facilitation and authoring instructional materials. Scott is an international keynote speaker and author of over 100 articles on software testing. He is a member of ACM, IEEE, American MENSA, the Context-Driven School of Software Testing and is a signatory to the Manifesto for Agile Software Development.

Copyright © 2009 PerfTestPlus, Inc. All rights reserved

Software Pedigree Analysis: Trust but Verify

By
Susan Courtney (susan@jli.com),
Barbara Frederiksen-Cross (barb@jli.com), and
Marc Visnick (marc@jli.com).



Abstract

As forensic software analysts, we are routinely called upon to review source code in the context of litigation or internal software audits. In our experience, it is extremely common for software developers to write software that uses or references third-party materials. These references may include source code incorporated directly into a program, source code routines that are statically linked as a part of the software, or the use of binary libraries that are dynamically referenced at the time a program is actually run. Although the inclusion of third-party materials appears prevalent in modern software development, in our experience, few companies fully document such usage and its associated licensing restrictions. This failure may expose these companies to unexpected legal liability. As a result, we conclude that quality software is defined not just by technical measurements, but also by the presence of a well-documented *pedigree* that itemizes all known use of third-party software and documents the license terms under which such materials may be used.

Quality pedigrees are established through prophylactic procedures, not reactive panic. “Best practices” include 1) carefully considered guidelines for using third-party materials, 2) thorough documentation of such use, and 3) the periodic performance of a forensic code audit, a procedure that uses a combination of automated analysis tools and visual code inspection to detect the presence of third-party material in a body of source code. Forensic code audits serve 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 three 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 done in the context of a merger or acquisition.

1 INTRODUCTION

During our professional careers, we have often observed software development from a vantage point that focuses on failures which may lead to litigation. Although these failures may take many forms, one of the most common is a dispute over ownership of the intellectual property embodied by software. A substantial number of these disputes focus on whether the software in question is derived from, or makes inappropriate use of, third-party materials. These materials may include, for example, source code examples incorporated directly into the comments of a program, source code routines that are statically linked into a program, or binary libraries that are dynamically referenced when a program is actually run. We have seen many cases where, during the course of due diligence or during the process of litigation, management is surprised to learn that their software violates license terms or copyrights of a third-party. This unfortunate situation often results from a combination of unsafe software development practices and a poor understanding of copyright or license conditions associated with third-party materials. As the custodians of software quality, quality assurance engineers are uniquely qualified to help mitigate this risk by enforcing quality processes that document software origins and protect the development effort from improper use of third-party materials.

In this paper we shall introduce the concept of software pedigree analysis by first discussing software ancestry, the legal risks that are attached to the use of third-party software, and the implications of current regulatory reporting requirements. We will then present best practices for establishing a quality pedigree and the concepts comprising a forensic code audit. We conclude the paper with some observations and lessons learned from performing hundreds of forensic code audits.

2 SOFTWARE QUALITY INCLUDES SOFTWARE PEDIGREE

2.1 What is pedigree?

Ponder these questions: If you were facing a lawsuit, or if your company was in the process of being acquired, could you demonstrate that you had implemented a set of “best practices” to guide your developers in using third-party material? Could you demonstrate that you documented all known instances of third-party material in your software, and that you complied with the terms of any licenses attached to that third-party material?

If you have your own staff of software developers, you might assume that these questions are simple to answer. That assumption may prove false if your developers have relied upon or incorporated any form of third-party software during development. In the event that they have, you must consider whether it is possible to identify all such third-party code, understand associated licensing restrictions (if any), and be able to demonstrate compliance with license terms.

If the origin of your software is called into question, can you produce records that prove when your code was designed and developed? Do those records identify all of the third-party code your developers used during the development process? Have you preserved a copy of each third-party license, and do your developers understand any restrictions placed on the use or re-use of third-party material? If you cannot authoritatively answer these kinds of questions about

your software, then you have failed to establish a pedigree sufficient to document your software's ancestry, and you may be putting yourself in legal jeopardy (or in an acquisition scenario, setting up your company or product for a reduced valuation).

2.2 Why should your company care?

While the tendency of developers to share and reuse computer source code is not new, the Internet and the proliferation of software distributed under open source licenses have made such sharing both easier and, from the legal perspective, more complex. In the authors' experience, it is common for developers to reference or incorporate third-party source code, giving little or no consideration to the potential legal consequences flowing from such use. In many cases, no formal record is retained to document the origin, use, or license terms associated with that code.

The failure to preserve such records can create a minefield of potential risks, legal and otherwise, possibly affecting both the software's valuation and your company's reputation.

Much of this risk comes from license terms that are attached to the use of third-party software. In particular, software that is distributed as *free*¹ or *open source*² software may be subject to a wide range of different use restrictions, whether under a *copyleft*-type³ license like the GNU General Public License (GPL) or the Mozilla Public License, or under *attribution*-type licenses such as the BSD license.⁴ Different licensing models place various conditions on both the current and *future* use of open source software. Any software that is derived from software under an open source license *inherits* these conditions; in turn, the licensing conditions are passed on to all subsequent generations of software derived from that software.

Open source software's potentially complex paternity suggests that it may pose a higher risk for certain types of legal exposure. Open source software is usually developed collaboratively via contributions from many different developers, and there is no sure way to guarantee that the contributors actually own the rights to material they are contributing. Further,

¹ Free is a matter of liberty, not price. The Free Software Foundation (FSF) offers the following definition of free software: "Free software is software that comes with permission for anyone to use, copy, and distribute, either verbatim or with modifications, either gratis or for a fee. In particular, this means that source code must be available."

²"Open source software (OSS) is defined as computer software for which the source code and certain other rights normally reserved for copyright holders are provided under a software license that meets the Open source Definition or that is in the public domain. This permits users to use, change, and improve the software, and to redistribute it in modified or unmodified forms." (http://en.wikipedia.org/wiki/Open_source_software visited June 25, 2009)

³ Copyleft is a play on the word copyright; it describes the practice of using copyright law to remove restrictions on distributing copies and modified versions of a work for others and requiring that the same freedoms be preserved in modified versions. A primary restriction of most copyleft licenses is that the user who incorporates copyleft materials must make available the source code of any derivative work they produce, under the same terms as the parent material's license.

⁴ There are now many distinct forms of open source licenses. The specific licenses mentioned in this paper are all available on the Internet. For example, the GNU General Public license (GPL) can be found at <http://www.opensource.org/licenses/gpl-license.html>. The GNU Library Public License (LGPL) can be found at <http://www.opensource.org/licenses/lgpl-license.html>. The BSD license can be found at <http://www.opensource.org/licenses/bsd-license.html>. The Mozilla Public License can be found at <http://www.mozilla.org/MPL/>. The Q Public License can be found at <http://www.opensource.org/licenses/qtpl.php>. The Apple Public Source License (APSL) can be found at <http://www.opensource.apple.com/license/apsl>. (All sites visited June 26, 2009.)

because of the dynamic nature of open source development, contributions that infringe a third-party copyright or patent are easily propagated and may contaminate multiple discrete software products before the infringement comes to light.

Code published under an open source license is not the only third-party code that your developers may have used without a full understanding of the license restrictions associated with that code. In some cases, developers maintain extensive collections of the programs they have written over the course of their careers and re-use handy bits of code over and over again when confronted with similar problems to solve. Ponder the ramifications of such re-use in the context of a consultant or employee who has worked in the past for one of your competitors. Here, the developer's decision to re-use portions of his prior work product may lead to copyright, patent, or trade secret litigation, even if you are unaware of the use of this material. In the authors' experience, this is an all-too-common scenario. The key to avoiding these pitfalls is a combination of education and internal control processes.

2.3 Managing the risk

Developers must be educated about the potential pitfalls of third-party code use.⁵ They should be given clear guidelines with respect to the requirements for recording the provenance and license terms of any code they contemplate using. A code review process should not only evaluate code quality but should also determine whether license restrictions are acceptable and consistent with corporate policies.

Additionally, companies should adopt a clear set of procedures that prohibit the use of materials which may have been produced for past employers, including source code, proprietary documents, and any other confidential information. Further, companies should provide developers with specific guidelines about *outbound* code, including contributions to open source initiatives, trade publications, and any other situation where a developer might contribute company-owned software or materials to a third party.

A company should communicate these guidelines and procedures to new developers and contractors during the indoctrination process and then periodically thereafter. A company should also ensure that during exit interviews developers and contractors are asked to turn over all company source code, and instructed that all code developed during their now-ending tenure is the intellectual property of the company and that it must not be used elsewhere.⁶

Companies may want to consider performing periodic software audits to make certain that corporate guidelines with respect to third-party materials are being followed. In many situations these audits can be incorporated into existing quality assurance processes. Audits by an outside party may be required in the context of litigation or acquisitions. In the event that undocumented (or unacceptable) materials are found, due diligence may require additional research and potential remedial action.⁷

⁵ This also applies to commercially licensed third-party material. We have observed numerous instances in which a body of code being purchased by a company contains materials under a third-party commercial license that is not readily transferable to the purchasing company. We have also observed instances where a programmer used a demo version of a commercially licensed product, without then obtaining a paid-up license for the product.

⁶ This restriction assumes that your employment or engagement agreements actively assert ownership of your employees' and consultants' work product.

⁷ Remediation can take a number of forms, such as removal of problematic material; complying with the terms of a third-party license (for example, providing appropriate attribution); or code refactoring (rewriting), possibly in the context of an independent software development project ("software cleanroom").

2.4 Source Code and SOX

Many companies, especially technology companies, categorize their source code as intellectual property (“IP”) on their financial reports. In 2002, in response to financial reporting abuses by major corporations, the Sarbanes-Oxley Act (“SOX”) was enacted by Congress of the United States of America. It placed into law new rules governing corporate accounting for public companies and enacted tougher ethical standards, including criminal penalties, for corporate officers for SOX violations. Intellectual property is not directly mentioned in SOX, but “part of the general requirements that come out of it are that companies are expected to monitor and disclose IP events, including the relationship between their intellectual property position and their financial performance.”⁸

For many companies, IP is a crucial part of its mission. However, the risks associated with IP are increasingly significant for all public companies, not just public technology companies. SOX is applicable to *any* assets that have a material impact on the financial condition of a public company, including material third-party IP encumbrances which may affect the ability of the company to conduct business.

SOX also requires corporate directors to certify that the company complies with all laws and contracts, including software licenses. Violating software license terms therefore places corporate executives at risk for personal liability. In this context, open source can be particularly problematic, since developers have access to open source materials outside normal software acquisition processes and may therefore incorporate open source into a company’s software without any review of its license terms.⁹

In light of SOX legislation, public companies are well served by establishing policies and procedures to bring structure and consistency to IP management. This includes identifying and assessing the materiality of IP assets. Further, companies should educate employees on the importance of IP assets, the procedures used for IP controls, and the importance of reporting to management material events related to IP assets.

3 HOW TO ESTABLISH PEDIGREE FOR A CODE BASE

3.1 Software Pedigree Analysis

Software pedigree is best established through use of systematic procedures and policies. “Best practices” include 1) carefully considered guidelines for using third-party materials, 2) thorough documentation of such use, and 3) the periodic performance of a *forensic code audit*. Forensic code audits help to ensure compliance with the corporate usage and documentation guidelines and to identify any unanticipated third-party use. The combined use of these three practices establishes a quality software pedigree. The maintenance of your software’s pedigree helps to manage the risk of unforeseen legal entanglements. A company using this type of

⁸ Quote from Leo Longauer in Barraclough, Emma “What Sarbanes-Oxley means for you” Managing Intellectual Property, 1 May 2008, available at <http://www.managingip.com/Article/1933503/What-Sarbanes-Oxley-means-for-you.html> [visited 6/25/2009].

⁹ “Taming The Open Source Monster” Open Source, 1 June 2006, available at <http://www.risk.net/public/showPage.html?page=331247> [visited 7/23/2009].

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.¹⁰

Software development “best practices” should include an approval process for the evaluation of third-party materials, as well as the review of supporting documentation regarding the origin of and license restrictions on those materials, before the materials are approved for use. A copy of the actual license text should be a required document for the review process and appropriately preserved for future reference.

Software pedigree documentation should be formalized and produced as an artifact of new development and updated during the on-going maintenance process. Until it is a common practice, it may be advisable to perform periodic compliance audits for this documentation. Developers (and managers) must be instructed to keep clear records related to the use of third-party code. These records should include a clear identification of the third-party code, including its specific version (if available), where it came from, how it is used, and what license terms apply to it. Additionally, the records should document which components incorporate the code and which components interact with it. Consider maintaining a special repository for approved third-party code and associated licenses and documentation, in order to preserve a clear record of the original third-party code base.

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.

3.2 Forensic Code Audits

Forensic code auditing looks for the presence of legally problematic material in a body of source code. In and of itself, a forensic code audit will not remediate problematic source code but serves as an effective means to detect problems, which can then be remediated in an appropriate manner. Forensic code auditing provides a proactive, objective review of source code to detect possible licensing problems. Essentially, a forensic code audit uses a combination of automated tools and visual inspection to determine if third-party materials have been incorporated into proprietary code. A variety of techniques may be used to identify:

- 1) literally-similar or near-literally similar incorporation; third-party code has been incorporated with no or little textual modification into code;
- 2) obfuscated incorporation; third-party code has been incorporated with textual modifications (e.g., variable and parameter name spoofing, function reorganization, etc.), but the textual narrative can be abstracted, compared, and shown to be similar;
- 3) textually dissimilar but logically similar incorporation; third-party code has been substantially rewritten on a textual level, but the original third-party logical flow remains intact.

¹⁰ Appendix One describes a software pedigree analysis done in the context of mergers and acquisitions. This paper focuses on audits done in the context of internal quality processes.

A forensic code audit has two purposes: 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). Be warned: determining ultimate origin and license conditions may present unexpected challenges. Sometimes the code may have an obvious third-party attribution but may come from a long-extinct source, i.e. a defunct “pre-Web” company who never posted any of its software or documentation assets on the Internet. Sometimes the code may be expressly distributed under a choice of licenses with no guidance as to what license applies in the immediate situation. In other cases, multiple licenses may apply due to the collaborative nature of the code’s development.

3.2.1 Scope the Audit

One of the first orders of business in performing a forensic 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. 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 more subtle evidence of copying. This scope determination must be made on a case-by-case basis, consistent with the circumstances triggering the audit, available time, risk tolerance, and cost factors.

3.2.2 Types of Analysis

Once the scope of the audit is established, the comparison can begin. Several types of comparison techniques may be used together during the analysis, including textual analysis, code comparisons, code scanning tools¹¹ (which usually involve some form of *digital fingerprinting*), and analysis of program logic.

Textual analysis examines a single body of code to determine whether it contains indications of third-party code in the form of copyright notices, license notices, or similar textual indicia.

Code comparisons match one body of code against another to identify areas of literal or near-literal similarity. Comparisons may be based on examining full lines of code, partial lines of code, significant internal names and literals, or some form of tokenized code representation.

Digital fingerprinting seeks to find identical files or sections of files by comparing algorithmically derived fingerprints of files or sections of files. The fingerprint usually takes the

¹¹ E.g. Palamida™ (<http://www.palamida.com/>), OSRM (<http://www.osriskmanagement.com/>), 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 (e.g. projects hosted on SourceForge), to minimally look for either in-file *snippets* of unattributed source code taken from third-party sources, or *digest matches* demonstrating that an entire file was taken from a third-party source. These tools are only as good as their search algorithms and fingerprint database and will not catch contamination from non-public sources (unless such materials are expressly incorporated into a custom fingerprint repository per customer request). In our experience, these tools must be accompanied by human analysis to weed out false positives and catch false negatives and, above all, to judge the contextual relevance of a *hit* on third-party materials.

form of a hash value.¹² (Appendix Two illustrates a simple use of MD5SUM hashes.) A reasonable analysis tool can compare the fingerprint of file A (or sections of file A) to a set of fingerprints as generated from file B and sections thereof (and file C, and so on).

In all cases, the more obfuscated the code, the more challenging the problem becomes for automated analysis. For example, simply changing the variable names in code copied from a third party may be sufficient to fool some automated comparison tools. Likewise, a minor amount of intra-function reorganization and/or name obfuscation may be enough to fool these tools. Automated tools can certainly facilitate the review process by highlighting potentially significant areas for review, but ultimately there is no substitute for direct visual inspection of two bodies of potentially matching source code.

The analysis of program logic and architecture is a more manually intensive and time-consuming process than the techniques above. Though automated analysis can still play a role in these situations, this scenario must rely upon visual inspection and the attendant experience of the examiner. While there are a variety of tools that permit an examiner to “walk through” a program in static fashion (visually examine the source code) and in dynamic fashion (watching the behavior of a program during its execution), these tools are usually directed to one body of code at a time and do not directly identify logical similarity. To date, that assessment is still best made by a human examiner.

3.2.3 Cautions

Given the enormous and ever-expanding universe of potential third-party material, no forensic software audit can provide a 100% guarantee that all possible third-party issues have been discovered in a code base. This is especially true regarding code that is derivative of, but nonetheless textually dissimilar to, third-party code.

In our 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 can not 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 or proprietary licenses.
- Archives within archives: In some instances, unexpected third-party subcomponents 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.
- SQL files and databases: 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 may be found in program documentation.

¹² MD5 (Message-Digest algorithm 5) is a widely used cryptographic hash function with a 128-bit hash value and is commonly used to check the integrity of files.

- Generated code: Some files may have been generated by a third-party tool, e.g. a GNU bison-generated parser file.¹³ These files 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. While in most cases we have determined that these instances are in fact benign referential citations, in some instances the citation was to material under a copyleft license, raising potential taint concerns (and/or possible publicity concerns).

This list is by no means exhaustive. The critical point is that a forensic code audit must take the *totality* of the materials into account.

4 EXPERIENCES

Over the course of several hundred forensic code audits, we have observed the following:

- Nearly every forensic code audit performed by the authors of this paper has turned up evidence of undocumented third-party material in the subject code base.
- A company’s disclosures regarding third-party materials have almost always proved incomplete, suggesting that many, if not most, companies have yet to implement effective pedigree tracking practices. While representations and warranties are certainly useful, our guiding axiom is “trust but verify.”
- 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.
- Automated analysis tools are useful, but the human brain is ultimately the best judge of contextual relevance.

5 CONCLUSION

Based on our experiences, we anticipate that many companies will, in their not-too-distant future, face some of the issues discussed in this paper. The growth of the open source movement and open source projects help support our premise, as more and more companies leverage open source materials to produce commercial applications. Regarding that growth, Amit Deshpande and Dirk Riehle (SAP Research, SAP Labs LLC) said:

¹³ 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 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.”

The significance of open source has been continuously increasing over time. [...] [T]he number of new open source projects, and the total number of open source projects are growing at an exponential rate. The total amount of source code and the total number of projects double about every 14 months.¹⁴

Additionally, in April 2008 Sun UK's chief open-source officer, Simon Phipps, stated that “[i]n 2005 we came to the conclusion that this was the future for the software industry.[...] We see the software industry switching over from its current delivery model to adopt far greater focus on open source.” He further stated that “most chief information officers simply don't have a policy for open source.”¹⁵

Given these and other considerations discussed in this paper, we believe that the importance of well-documented software pedigrees and carefully-considered usage guidelines cannot be overstated. Companies using quality pedigree programs will be better positioned to deal with licensing questions, respond to merger and acquisition scenarios, or act quickly in the event of a lawsuit.

¹⁴ Proceedings of the Fourth Conference on Open Source Systems (OSS 2008). Springer Verlag, 2008. *Also available at* <http://dirkriehle.com/publications/2008/the-total-growth-of-open-source/> [visited 7/16/2009].

¹⁵ <http://www.zdnetasia.com/insight/software/0,39044822,62040810,00.htm> [visited 7/17/2009].

6 APPENDIX ONE -- SOFTWARE DUE DILIGENCE

A software pedigree analysis done in the context of mergers and acquisitions is often referred to as Software Due Diligence (“SDD”). Like a software pedigree analysis, it is a multi-step process that is used to determine the true origin of third-party material and to identify the applicable licenses associated with that material. The steps of the process include the scope determination, the preparation and delivery of the materials, and the preparation of a budget for the effort. Ideally, SDD involves independent analysts and possibly outside counsel since the buyer should not see the seller’s actual source materials, only a summation of findings based on that material. If the transaction should happen to fall through, an independent analysis framework helps avoid litigation related to misappropriation claims.

SDD is not always an easy assessment. There may be multiple sources for particular materials, with conflicting licenses and ultimately ambiguous origin. Often there is a single source for the code, but there may be a choice of licenses associated with the code. It may be unclear under which license the code is distributed.

SDD analysis is usually focused on literal expression in the code, i.e. obvious third-party attribution notices, suspect words or phrases, textually-similar code and file-level similarity. While non-literal analysis is possible, it is usually labor-intensive and therefore time-consuming and costly.

A SDD effort must be scoped and sized with input from all of the relevant players. Deliverables generally include the performance of the analysis, a report on the findings and, if necessary, a remediation review. The scope of a SDD is usually decided by the factors of both time and budget. A detailed review is preferred but not always possible. Time and/or cost pressures may limit analysis to obvious third-party materials. The scope is thus a factor of risk tolerance. An abbreviated assessment may be useful, but be riskier in the long run.

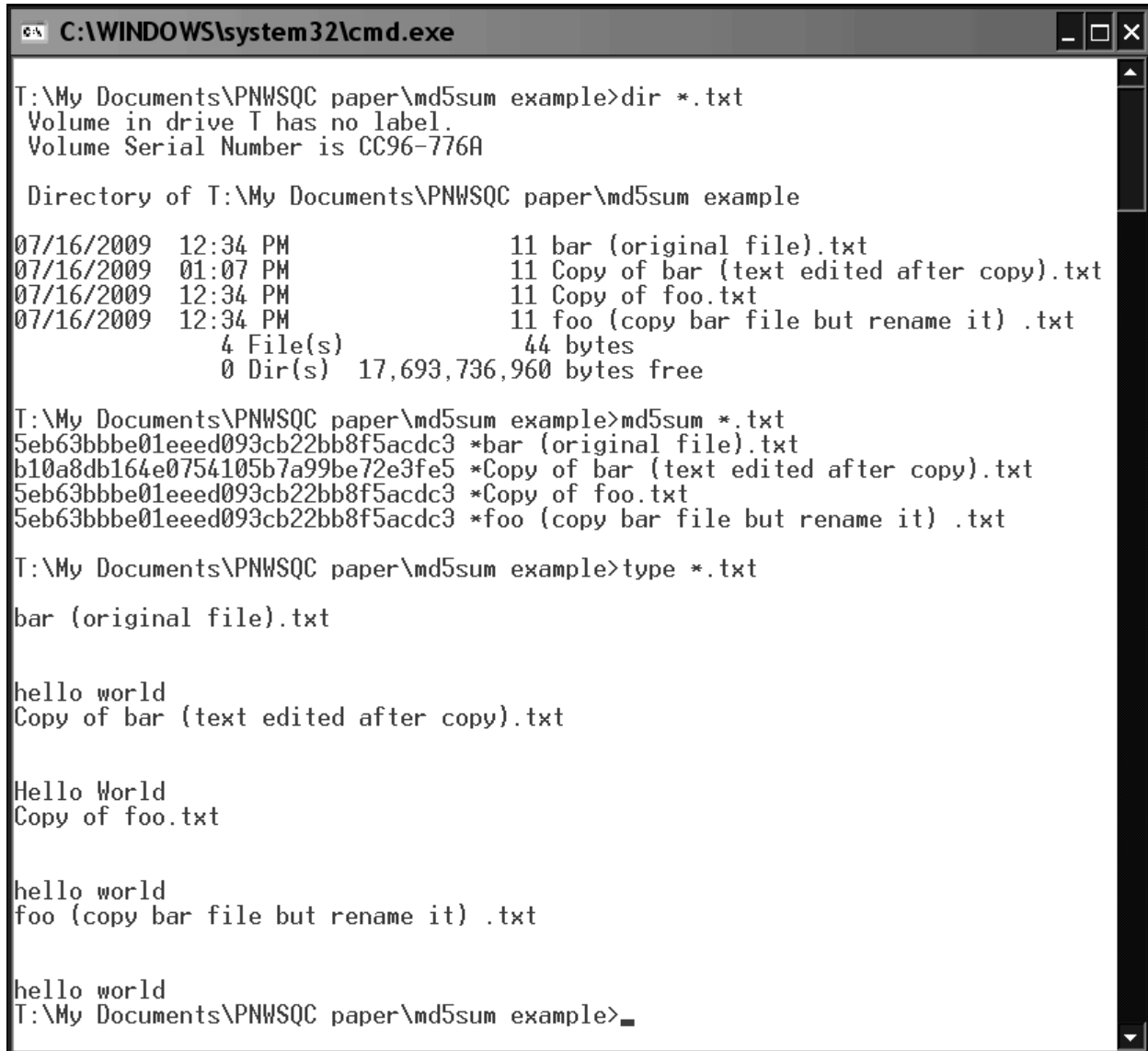
The third-party source code universe is vast and still largely uncharted. SDD cannot provide a 100% guarantee that all third-party code, derivative or otherwise, has been identified in a body of code since that is both a difficult and complicated task.¹⁶

In the end, most problems identified in the report findings are remediable. Some problems are better resolved prior to the close of the sale (e.g. issues related to continuing support obligations for acquired company’s software) while other can be resolved post-close (e.g. the disposition of non-critical source code). When core issues related to software are not easily resolved, it may be time for a reality check. How much liability do you want to buy?

¹⁶ Further, patents are the elephant in the room; i.e. even if the pedigree of the software is well established, the software could still infringe one or more patents.

7 APPENDIX TWO -- MD5SUM EXAMPLE

The simple example below illustrates how the values computed for md5hash sums relate to the file contents and not to other considerations such as file name and date created. The original file “bar (original file).txt” was copied into three different files. It was edited once in the file named “copy of bar (text edited after copy).txt” times; three of the four differently named files have identical md5sum hashes. The edited file has a different *fingerprint* i.e. it was changed and its md5sum hash reflects that change.



```
C:\WINDOWS\system32\cmd.exe

T:\My Documents\PNWSQC paper\md5sum example>dir *.txt
Volume in drive T has no label.
Volume Serial Number is CC96-776A

Directory of T:\My Documents\PNWSQC paper\md5sum example

07/16/2009  12:34 PM                11 bar (original file).txt
07/16/2009  01:07 PM                11 Copy of bar (text edited after copy).txt
07/16/2009  12:34 PM                11 Copy of foo.txt
07/16/2009  12:34 PM                11 foo (copy bar file but rename it) .txt
               4 File(s)              44 bytes
               0 Dir(s) 17,693,736,960 bytes free

T:\My Documents\PNWSQC paper\md5sum example>md5sum *.txt
5eb63bbbe01eeed093cb22bb8f5acdc3 *bar (original file).txt
b10a8db164e0754105b7a99be72e3fe5 *Copy of bar (text edited after copy).txt
5eb63bbbe01eeed093cb22bb8f5acdc3 *Copy of foo.txt
5eb63bbbe01eeed093cb22bb8f5acdc3 *foo (copy bar file but rename it) .txt

T:\My Documents\PNWSQC paper\md5sum example>type *.txt

bar (original file).txt

hello world
Copy of bar (text edited after copy).txt

Hello World
Copy of foo.txt

hello world
foo (copy bar file but rename it) .txt

hello world
T:\My Documents\PNWSQC paper\md5sum example>
```


8 APPENDIX THREE -- PEDIGREE CHECK LIST

- What third-party code is in use?
- Which version?
- Where did it come from?
- When was it procured?
- What license(s) apply to the third-party code?
- Which version of the license?
- Is a copy of the license on file?
- What is the scope of intended use 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?
- Are all license compliance requirements related to the third party code met?
- What processes are in place to track compliance issues?
- Who is responsible for compliance?
- Who is responsible for code review?
- What directives have developers been given with respect to this code?
- Have developers affirmed they followed directives?
- What processes ensure developers follow the directives?
- 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 you plan to perform periodic compliance audits?
- If so, who will perform the audit?
- How will the audit results be documented?
- 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?

BIOGRAPHY

Susan Courtney is a forensic software analyst and has worked as a consultant for Johnson-Laird, Inc. She has done forensic software analysis for patent and copyright cases, performed electronic evidence analysis, and assisted with data preservation and discovery. Ms. Courtney is also the owner of SLC Software LLC, a software consulting firm that provides a variety of software-related services including business systems analysis, quality assurance and project management.

Barbara Frederiksen-Cross is the Senior Managing Consultant for Johnson- Laird, Inc., in Portland, Oregon. Barbara is a forensic software analyst specializing in the analysis of computer-based evidence for copyright, patent, and trade secret litigation. She is also an expert in computer software design and development, the recovery, preservation, and analysis of computer-based evidence, and computer systems' capacity issues. Mrs. Frederiksen-Cross was appointed as Court Data System Advisor to the Honorable Marvin J. Garbis, in the U.S. District Court for the District of Maryland in December 2000.

Marc Visnick is a forensic software analyst and attorney based in Portland, Oregon, and a senior consultant with Johnson-Laird, Inc. He specializes in forensic software analysis for patent, copyright and trade secret litigation, as well as software due diligence, independent development project design and supervision, and electronic evidence preservation, recovery and analysis. Mr. Visnick is past Chair of the Oregon State Bar Computer and Internet Law Section.

Reconfiguring the Box:

Thirteen Key Practices for Successful Change Management

Leesa Hicks
leesa.hicks@tek.com

Abstract

Existing processes help organizations work in an organized and predictable way, but they also provide resistance points for improving how they work. Tektronix “standardized” on IBM Rational ClearCase for configuration management years ago, but because it had no built-in process automation, each product team developed their own customized processes and automation for using ClearCase.

Ironically, the lack of process automation in the early versions of ClearCase was actually part of its initial appeal. However, both ClearCase and development organizations have matured greatly since then, and ClearCase now provides automated processes with Unified Change Management (UCM). More recently, Tektronix adopted IBM Rational ClearQuest for change tracking, which adds yet more value when integrated with ClearCase UCM.

Even with all the new capabilities and automation that UCM provides, motivating development teams to change how they use ClearCase has been challenging, in spite of the issues they have been experiencing with their existing ClearCase usage. They still need to be convinced that the value added by adopting UCM outweighs the cost of changing how they work, even though they are using the same tools. This paper describes how we have helped development teams break out of the constraints of their existing configuration management processes and improved their development processes with successful adoptions of UCM, including thirteen key practices used to bring about these beneficial changes.

Biography

Leesa Hicks is a principal software engineer in the Software Engineering Services group at Tektronix Instruments in Beaverton, Oregon. She has worked in a number of roles in software engineering throughout her career, including: software developer, project lead, technical lead, technical specialist, and manager. Leesa has also worked in a number of industries as well: banking, manufacturing, IC design automation, automated test equipment, software sales support, and IT. A common thread through all these roles and industries has been her work in process improvement. Leesa is currently responsible for ClearCase and ClearQuest strategy and development and ClearQuest operations at Tektronix.

Leesa has an M.S. in Computer Science from the University of California at San Diego. While at Tektronix, she has been nominated for a Technical Excellence award, and she has received two Customer Excellence awards.

Copyright Leesa Hicks August, 2009

Background

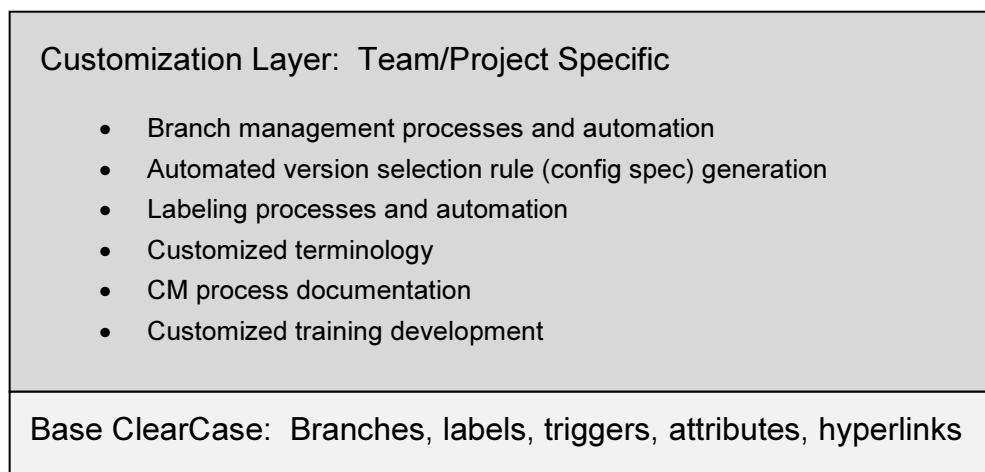
IBM Rational ClearCase and ClearQuest are the corporate-supported configuration management and change tracking tools at Tektronix. As a member of Software Engineering Services group, I work with teams to resolve issues they are having with using these tools. These interactions provide opportunities for introducing teams to Unified Change Management (UCM) when they have significant problems that can be solved by using UCM. The following sections provide a brief comparison of the effort required to adopt Base ClearCase versus UCM.

Configuration Management

There are many definitions of configuration management available from numerous sources. This paper concentrates on configuration management in terms of version control, configuration identification (baselines or labels), change process management, development process management, build and release management, defect and change tracking, and change status reporting.

Base ClearCase

When Tektronix first deployed ClearCase, it provided a set of mechanisms for managing concurrent, distributed development, but no automation. This is referred to as *Base ClearCase*, and using it requires developing a significant customization layer on top of ClearCase. It typically takes three to six months for a team to develop the initial customizations needed to adopt Base ClearCase, requiring maintenance and enhancements as needs change. The following diagram illustrates the types of automation and infrastructure necessary to use Base ClearCase. Note that this effort is not generally transferrable between teams, and possibly even between projects.



As a quick example, in order to work on a branch in ClearCase, you must create a branch type, and then you must create a configuration specification (version selection rules) to make changes on that branch. If you are supporting this for a team, common customizations include developing some type of branch management policies and processes, automating branch creation to support the team's development processes, automating the generation of configuration specifications, automating merging between branches to integrate work together, developing labeling policies and processes, and automating the creation of label types and label creation to support the build and release process.

Because Base ClearCase customizations tend to be unique between teams, there is a great deal of overhead that is repeated throughout the enterprise. Each customization incurs support overhead, and product updates can break these customizations. In addition, when developers move from one development team to another, they need to learn the new team's processes and terminology for using ClearCase. This can be significantly different from their previous team's processes, and may even rival the overhead of learning a completely different configuration management tool.

Unified Change Management (UCM)

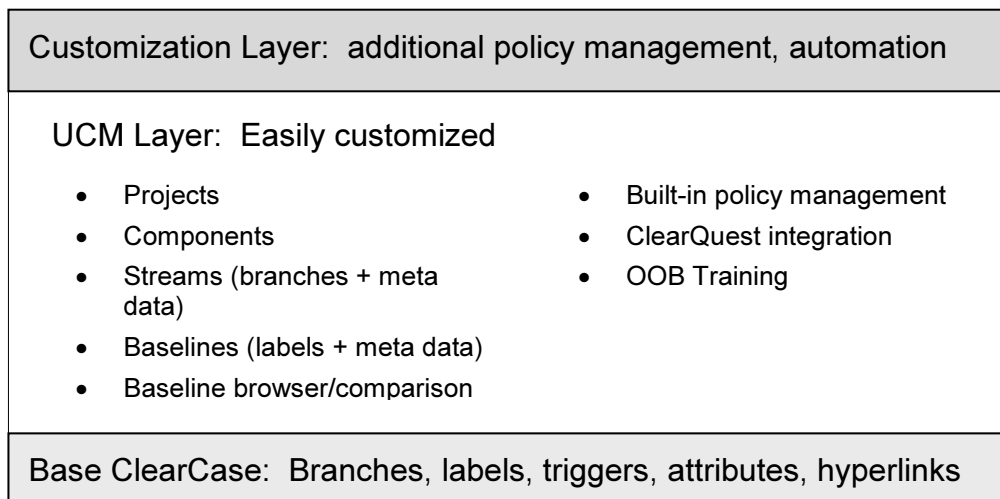
UCM is a layer on top of ClearCase that automates most of what is needed to use ClearCase out-of-the-box, and it provides more flexibility with organizing and managing development work. UCM models configuration management in familiar ways for development teams, using projects to organize significant development efforts, components to organize sets of related files versioned as a whole with baselines, and activities to track change sets for each task a developer works on.

Components generally consist of the significant architectural pieces of a system, and they are a key mechanism for providing flexibility with organizing development. Loosely-coupled components make it possible to define projects using producer-consumer relationships, for example, rather than the classic release model that treats all the files comprising a system as a single entity or configuration.

For example, one UCM project may be used to modify the user interface component, with read-only access to the components required to build it, and another project may be used to modify a component or set of components that the user interface is dependent upon, but which does not need access to the user interface component itself. This makes it possible for the user interface team to manage its own development, integration, testing, and releases of the UI, independently of the other project, and vice-versa. This flexibility is useful when release cycles differ between a system's architectural pieces or components, and any case where specific team members work exclusively on one portion of a system.

UCM can also be integrated with IBM Rational ClearQuest, which links each UCM activity in ClearCase to an actual defect, enhancement request, or any type of change request in ClearQuest. ClearCase information and operations are available from ClearQuest and vice-versa. For example, developers can view the change set for a defect (the list of file versions modified to fix that defect), and they can view the changes they made to any particular file to fix that defect. This integration also provides additional capabilities for change status reporting, which is particularly useful for build management and project status evaluation and reporting.

The following diagram provides an illustration of the types of automation that UCM provides, and there are many ways to customize it with built-in mechanisms. There may still be a need for some additional automation on top of UCM, but this is minimal compared to Base ClearCase.



With UCM, there is much more commonality between different teams and projects, so the overall overhead of using UCM throughout the enterprise is greatly reduced. Developers can move from team to team and project to project using UCM much more easily as well, because they only need to understand how this team or this project uses UCM rather than how this team or this project uses a completely customized implementation of ClearCase.

Configuration Management Advancements

This section describes the journey of three teams who have adopted UCM at Tektronix, including their challenges and requirements, as well as the results they have achieved with UCM.

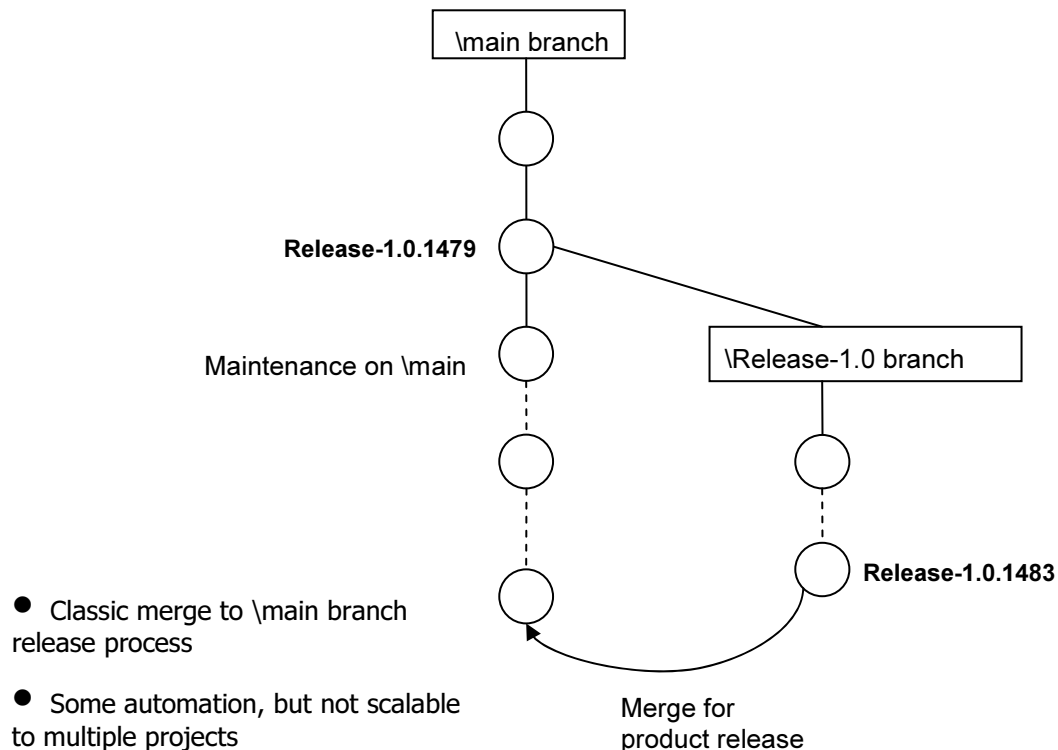
Red Team

The Red Team was the first team to adopt UCM at Tektronix. This team is often an early adopter of newer methods and tools, and they recognized the value of using UCM without the benefit of any other team's experience. They needed to change how they were using ClearCase primarily because they needed to change from working on one release at a time to working on multiple releases concurrently, although they had other issues as well. Here is a summary of their challenges and requirements:

Table 1: Red Team's Challenges and Requirements

Geographically distributed development team: ~20 Oregon team members, ~10 remote team members.
Developers need to work in isolation
Performance issues with merging, applying labels, opening the version tree browser
Difficult to identify changes made for a task
Needed to change from working on one release at a time to working on multiple releases

The following diagram shows the Red team's original branching process, which supported working on one release at a time while maintaining the existing release. This process would not scale to support working on multiple releases concurrently:



I started working with the Red Team several months before they needed to start working on multiple concurrent releases. Our options were for me to help them switch to using UCM, or they would need to develop some new processes and automation for using Base ClearCase. I worked with the key

stakeholders to understand their needs (see Table 1), and then provided some training and demonstrations of UCM. They did not have time to meet with me more than once a week, so we met weekly for an hour over seven months discussing UCM, identifying their use cases, determining whether UCM would meet their needs, and then planning the actual UCM deployment. Because they had so little time to work on this, I developed the entire configuration management plan for them, but I would have preferred that this be more of a joint effort.

Use of the ClearQuest integration was the only portion of the adoption that made the Red Team nervous, because UCM requires that you associate every change made with a ClearQuest activity. They were used to using ClearQuest to track defects and enhancement requests, but they would also need to track everyday tasks as well. There was no option here other than to provide assurances that the integration would work well for them.

In order to minimize the impact to the team and to manage the effort required by Software Engineering Services, we completed the UCM deployment in phases. Their deployment required restructuring their source code repositories, so we did this prior to their actual UCM adoption. We provided formal training the week before the UCM deployment, implemented the UCM configuration over a weekend, and I sat in their work area for the first two weeks of their UCM adoption to provide direct support.

The switch to UCM went rather smoothly for this team, and here is a summary of their results:

Table 2: Red Team's Results

Met their goal with developing multiple concurrent releases. Have supported as many as six concurrent programs
Some initial pushback from developers on the activity based tracking with ClearQuest, but they quickly came to see the benefits. No more manually recording what they had changed
No more version selection rule (config spec) confusion, since these are auto-generated by UCM
Remote team able to integrate work into the project with UCM remote deliveries. → Eliminated nightly merge scripts and manual conflict resolution for remote development
Eliminated Base ClearCase customizations – reduced script maintenance
Improved performance → UCM rebase and deliver (merge operations) much faster than custom scripts → Reduced nightly build time by ~20 minutes with incremental baselines
Have more options for sharing work

As you can see from Table 2 above, the Red Team's UCM adoption was a clear success for them, and they quickly concluded that it was the right thing for them to do. As predicted, they even appreciate the ClearQuest integration in spite of their initial apprehension. They have also become strong advocates for its usage, and they have been very helpful in assisting with other teams considering adopting UCM.

Purple Team

The opportunity to work with the Purple Team came from a change agent, Doug, who attended a presentation of the work we did for the Red Team. Doug obtained management's agreement to perform an assessment of whether UCM would be a good solution for their configuration management needs. Here is a summary of the Purple Team's issues and requirements:

Table 3: Purple Team's Issues and Requirements

Geographically distributed development: ~26 in Oregon, ~26 in India
Team members at each site use different configuration management processes. The Oregon team uses a home-grown DevRel automation system with ClearCase, but the India team is unable to use DevRel because it was designed for a co-located team. Neither group understands the other's configuration management processes.
DevRel is supported primarily by one individual, with one backup, requiring ~12 hours per week to

maintain (this primary individual has since left the company)
DevRel tracks change sets for Oregon team only; India work contributed by many developers is merged in as a single change set
Need to reduce the overhead of managing concurrent, distributed development
Developers need to work in isolation and project work needs to be managed in isolation
Oregon responsible for integrating India work into the overall project, even though most of the team members were in India, ~3-5 hours per week overhead in Oregon
Need to be able to determine what is being released. Sometimes defects fixed in one release are lost in subsequent releases

Doug solicited input from the Red Team to learn why they switched to UCM and what issues they have with using it. The Red Team communicated that they couldn't manage concurrent releases without having adopted UCM, and the only negative thing they had to say was that the developers initially didn't care for having to track all their tasks in ClearQuest, but that they had quickly adjusted. One of the comments from the Red Team was that it had been years since they had looked at a config spec, which for anyone who has used Base ClearCase, understands the difficulties with creating and maintaining config specs.

After considering the benefits and the cost of changing, the Purple Team decided to deploy UCM in phases. At the time of this writing, we have completed some preparatory work for adopting UCM for the entire Purple Team, and we have completed the first UCM adoption for one of the development teams within the overall group (phase 1). This team has ~16 members in India and ~4 in Oregon. Although most of the team works in India, all of the project integration work was originally being done in Oregon.

Doug adapted the Red Team's configuration management plan as a starting point for the Purple Team, and one of the Oregon software leads, Byron, pushed for this phase 1 team to adopt UCM. Byron saw value in having the ability for this team to manage their own project integration, and recognized UCM as having value for implementing a well-defined configuration management process.

I worked with Byron and key stakeholders in India to guide them through the adoption process similarly to what I did for the Red Team: education, demos, deployment decisions, etc. This was more challenging than for the Red Team because of the time differences between India and Oregon, as well as language and cultural differences. We provided training for the team just prior to the initial adoption, and we deployed UCM for this phase 1 team over a weekend, including re-structuring their source code repository.

With the phase 1 team's UCM adoption, we have accomplished the following:

Table 4: Purple Team's Phase 1 Results

The entire phase 1 team now uses the same configuration management processes
The India project leads now manage project integration and releases independently of Oregon
Oregon team members are able to contribute their work using built-in support for geographically distributed development
Change sets for each activity are tracked for all team members
Have the ability to determine what is in a release/baseline
Provides additional control over the integration, test, and release cycle

This adoption has also been successful, as you can see from Table 4, although it is fairly recent and the developers are going through the normal pushback on having to track all their tasks in ClearQuest. I expect that they will quickly get used to this, just as the Red Team did. Software engineering management is also happy with the results and is committed to adopting UCM for the remainder of the Purple Team later this year.

Green Team

The Green Team was experiencing some significant performance issues with using ClearCase, and they had started investigating options for switching to a different configuration management tool. I had an opportunity to explore their issues with two of the team leads and discovered that it was their own customization of ClearCase that was causing most of their performance issues. They were open to considering UCM, and here is a list of the Green Team's issues and requirements:

Table 5: Green Team's Issues and Requirements

Geographically distributed team
Partially committed changes accessible on the \main branch
Excessive overhead with merging files between branches
Difficulties integrating the remote team's work into the overall project
Need to reduce the configuration management support burden

Once I was able to educate the team leads on how UCM could solve these issues, they were willing to consider UCM but they were still skeptical. I completed some performance tests to compare to what they had done with Base ClearCase and Subversion, and this helped convince them that UCM was worth trying. I developed a configuration management plan for them, and in this case we deployed UCM for a new development project rather than re-configure work already underway.

Unlike the Red and Purple Teams, the Green Team prefers to use the command line for using ClearCase, and they were not interested in integrating ClearCase UCM with ClearQuest. They also don't care about UCM change set tracking per se, only in that it helps them check in files that they have modified for a task and automates merging. They had fixed ideas for how they wanted to work, so it was challenging at times to try to present other points of view and other things to consider. They prefer to write a script, for example, to filter out noise in textual output, rather than use a graphical user interface that already does this for them, as well as other benefits.

One thing that I was unable to convince them of was to take advantage of the remote development support that is built into ClearCase UCM. When they were using Base ClearCase, they had a lot of difficulties with merging in work done by the India team members, and this is admittedly a challenge with Base ClearCase. However, they insisted that the India team use the ClearCase remote client so that they work directly in the source code repository hosted in Oregon. Recently, this has become an issue due to WAN connectivity and bandwidth problems causing performance issues in India, so India will be changing to use native clients and the remote development support in UCM as I had originally tried to recommend.

Here is what we have accomplished with the Green Team:

Table 6: Green Team's Results

Geographically distributed development support
No more partially committed changes on the integration branch
The overhead of identifying what to merge was reduced from 20+ minutes to seconds
Individual developers now responsible for integrating work into the overall project; eliminated difficulties with integrating remote team's work
Software engineering services now providing most of the ClearCase support

The Green Team was trying to avoid defining aliases and writing scripts as wrappers around ClearCase operations, but they have implemented some of this for UCM, anyway. With their Base ClearCase usage, they had to reproduce any problem they were having outside of their ClearCase customizations before being able to seek support, which was rather time consuming for them and understandably frustrating. With UCM, they are able to obtain support from Software Engineering Services as needed, which is a very welcome improvement for them.

The Green Team has stated that they can't imagine how they managed development before adopting UCM, and they will be using it for all new projects going forward.

Adoption Cost Summary

The following table summarizes the effort required by Software Engineering Services to deploy UCM for the three adoptions. As you can see, the effort required to adopt UCM, even for the first team at Tektronix, is significantly less than the typical 3-6 month effort required to adopt Base ClearCase.

Table 7: Adoption Cost Summary

Category	Red Team	Purple Team	Green Team
Relative complexity	High	Medium	Low
Education, planning, and documentation	8 days	5 days	2.5 days
Source repository re-structuring	14 days	1 day	.5 days
Configure UCM	1 day	.5 days	.5 days
Deliver Training	2.5 days	2.5 days	2 days
Post-deployment support	7 days	5 days	5 days
Totals	32.5 days	14 days	10.5 days

Note: The Red Team's UCM adoption was the first one within Tektronix, so there was more effort involved overall to develop some basic infrastructure. We have been re-using plans, documentation, tools, etc. to reduce the effort for subsequent adoptions.

Thirteen Key Practices for Success

This section summarizes the primary activities and approaches that have made these UCM adoptions both possible and successful. Many of these topics will be familiar to anyone who has experience with trying to change how people work. Even though this paper focuses on improving configuration management processes, these practices apply no matter who is involved or what improvements are needed.

Listen and Understand

Since I don't work in the development groups directly, it is extra challenging to initiate changes in the way the development groups use ClearCase. What do I know about their problems? It doesn't matter to them that I have worked in software engineering my entire career. It doesn't matter to them that I have worked with ClearCase for over 15 years. It doesn't matter to them that I have years of experience with helping teams adopt ClearCase/ClearQuest/UCM. What gets their attention is whether I really understand their needs and their problems.

When I have the opportunity to work with a team, the first thing I do is have them tell me what their issues are, what they would like to see happen, and then repeat back to them what I think I heard. I don't do anything else until they see that I understand how they currently use ClearCase, as well as understanding the problems they need to solve. This is essential to establishing a good working relationship with the team.

Seek out Early Adopters

When I started at Tektronix, all the development teams were using Base ClearCase, and they didn't seem interested in changing how they use it. It is always difficult to change how people work, and this doesn't vary no matter how beneficial the changes are. Getting that first team to change is always the most difficult to accomplish, because everyone wants a successful example of a UCM adoption at the company to help in their decision making process.

So how do you get that first team? For anyone experienced with promoting process improvement, and consistent with my experience, this has always been an early adopter: individuals or teams that

understand the benefits of changing how they work compared to the cost of changing. In this case, two things needed to happen, a team whose configuration management requirements were changing, and a team that was able to understand that UCM could satisfy their requirements.

One Successful Adoption Leads to Others

Since the Red Team has adopted UCM, it has been much easier to discuss a possible UCM adoption with other teams. When the Purple Team was considering UCM, they met with two of the Red Team members to learn about their experience, and they recommended UCM wholeheartedly. The Red Team's software lead has been a continuing source of help with other teams as well. He provided a copy of their build scripts for the Purple Team to use, and was willing to help adapt them for the Purple Team's needs.

At the time of this writing, we have two additional teams considering a UCM adoption. One team has been asked to switch to UCM by the Purple Team's software engineering manager after their successful pilot adoption with phase 1. Another team is planning to adopt UCM because their team members have used it either at Tektronix or other companies.

Prototyping

Prototyping has a number of uses, including: 1) testing the new environment and processes prior to deploying them to verify that they will work, 2) developing any tools or automation in preparation for the adoption, 3) testing the conversion process. Prototyping is particularly important when a team is nervous about whether the planned changes will actually work for them.

We used prototyping for the Purple Team's phase 1 adoption to develop a build script for UCM, starting with the Red Team's UCM build script. The Purple Team's original build script only worked with their customized DevRel system. Developing and testing the new UCM-based build script prior to the adoption helped ensure a smooth transition to UCM.

Pilot Program

Using a pilot program to reduce the risk of changing is also a useful tool. We did this with the Purple Team's phase 1 adoption because they were still nervous about switching to UCM, and this was a good way to reduce the problem size and solve some significant issues for them. We reduced the team's size from ~52 to ~22, the number of components from ~10 to 4, with a corresponding decrease in the number of projects. Since this phase has been successful for the phase 1 team, the Purple Team is planning to adopt UCM for the remainder of the team later this year.

Work at the Pace of the Adopting Team

Development teams at Tektronix have a policy of changing tools, methods, processes, etc. only in between projects or releases to minimize schedule disruption. When I work with a team on a UCM adoption, we work out what their target date, and then we determine whether we can meet that date. If we can't meet that date, it will be weeks or months before the next opportunity comes up for that team, depending on the size of their development iteration or how close they are to a release. We do everything possible to meet their first target date; if this is not possible, we work out a later date and make sure to meet that date.

Education

After verifying my understanding of the team's requirements and issues, I spend time explaining and demonstrating how UCM works to their key stakeholders. I spend as much time as the team needs to develop sufficient understanding. These individuals will be deciding whether UCM will solve their problems, and if so, how to customize it for their needs. I only educate the team on what they need to know, however. There are many technical details that are important for the deployment infrastructure but are not relevant to how the team will use UCM, and so I don't share these details with the teams.

This educational process is a critical piece of planning a successful deployment, both for understanding what to expect, but also for helping the rest of the team understand why they are changing how they use ClearCase. When developers complain about having to track activities with UCM, the key stakeholders can articulate why this is valuable to the project team.

Allow Time for Absorption

This is related to the initial education process, and there are two aspects to this. First, it is important to give the key stakeholders time to absorb how UCM works prior to making decisions about how to customize it for their teams. There are a number of terms and concepts that are new in UCM, and it is important to understand these, to understand what is possible in UCM, as well as understanding its constraints. When the stakeholders start asking about usage scenarios with UCM, this tells me they understand the concepts, and this is where the planning process actually begins.

Secondly, the entire adopting team needs time to adjust to the coming changes. Allowing sufficient time for everyone involved to absorb the plans to move to UCM, including the reasons for changing and how and when it will be accomplished, significantly reduces resistance to the changes.

Help Solve Real Problems

UCM is not a panacea, nothing is. It is not the answer for every configuration management use case. Honesty is required for establishing and maintaining good working relationships with the development teams, and being honest about what UCM can and can't help with, or shouldn't be used for, is an important for solving the appropriate problems.

Practice Detachment

I enjoy helping a team solve problems, because I obtain a lot of satisfaction with helping people work in more efficient ways. It doesn't matter to me whether that is within my own group or with other development teams, it is all rewarding. However, it is important not to get attached to the outcome, but instead focus on a supportive role and help lead the teams to their own solutions. Some teams need more guidance than others, and so I try to match their needs in this regard.

Sometimes it is necessary to agree to a solution that you know is sub-optimal, because the team needs to experience for themselves how it will work out. This occurred with the Green Team when they insisted on a single-site solution for their distributed team; they based this on their experience with Base ClearCase and were not open to a distributed solution with UCM. If I were attached to what I considered was the optimal solution, I could have pushed for the distributed solution and possibly damaged my relationship with the team. Instead, after working with their single-site solution for many months, they came back to me when they needed to resolve the issues that were predictable from my perspective, but not from theirs, and we are working together to deploy a distributed solution.

Planning, Planning, Planning

As with most activities, sufficient planning is a key success criterion, and a UCM deployment is no exception. A UCM deployment plan includes the requirements and issues to be resolved, the composition of the team, responsibilities, the project and stream policies to be used, and all the major tasks involved in completing the adoption. When the effort is large enough, the deployment is organized into manageable phases. Since creating the initial deployment plan for the Red Team, we have been able to re-use large sections of it for subsequent adoptions, thereby reducing the overall planning effort required for each deployment.

The UCM deployment plan has also served as a reference for teams considering a UCM adoption, so they can gauge the effort involved and as assurance that Software Engineering Services will provide adequate support for the deployment.

Training

It is critical to adequately train the entire team on how to use UCM prior to the adoption. We were able to use general UCM training for developers from an external vendor, and then we added 1-2 hours for developers for their team's UCM processes, plus 2-4 hours for project integration training for project leads.

Training is the one aspect of UCM adoptions that Software Engineering Services insists upon because the Tektronix culture eschews training - this is a case where practicing detachment is inappropriate. It is only possible to have a successful adoption with UCM if the developers and project leads are properly trained. Sufficient training also minimizes the support burden for Software Engineering Services, a cost

that is easy for development groups to discount because it is not visible to them, but it is essential for ensuring adequate support resources for the development teams.

Post-Adoption Support

For each team's UCM adoption, we have dedicated a Software Engineering Services support person to physically sit in the adopting team's work area for the first week or two of the initial adoption. The actual time we dedicate is up to the adopting team. This eliminates the usual overhead of requesting support and waiting for a response, and has proven to be highly valuable in avoiding frustration during the initial adoption.

Summary

To date, we have completed three successful UCM adoptions at Tektronix, which has eliminated three sets of customized Base ClearCase infrastructure and helped teams use ClearCase more efficiently and effectively. As of this writing, we have two additional teams considering UCM adoptions, and the Red Team is advocating using UCM for managing code that is shared amongst several teams.

The key practices described in this paper were instrumental to the success of these UCM adoptions. I have learned these practices throughout my career using others' experiences and wisdom with process improvement efforts, as well as learning from my own failures. Of course I also have a wealth of technical skills that I utilized to ensure that UCM is used and deployed using best practices. However, the most difficult aspect of working on these adoptions has been convincing teams of their value in helping solve their problems. Based on feedback from the adopting teams and the current demands for more adoptions, we have been highly successful with advancing configuration management practices using these thirteen key practices.

Leveraging Code Coverage Data to Improve Test Suite Efficiency and Effectiveness

Jean Hartmann
Test Architect, Developer Division Engineering

*Microsoft Corp.
Redmond, WA 98052*

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

Biography

Currently, a Principal Test Architect in Microsoft's Developer Division with previous experience as Test Architect for Internet Explorer. My main responsibility includes driving the concept of software quality throughout the product development lifecycle. 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.

Abstract

During each release of Visual Studio, substantial time and resources are expended in test case development, execution and verification. Thousands of new tests are added to existing test suites without any kind of review regarding their unique contribution to test suite effectiveness or impact on test suite efficiency. In the past, such unbridled growth in test collateral was sustainable without significantly impacting product release, offset by simply increasing machine and staff resources. With the growing number of test configurations in which these tests need to be run, this is no longer feasible – it is time to clean up!

In this paper, we describe how we leverage existing code coverage data, together with reduction techniques, to help each test team analyze its test suite and guide them in improving its effectiveness and efficiency. The analysis focuses on identifying groups of tests cases given specific tactical goals – for example, increasing current test suite stability and reliability, assisting with test migrations, reducing test suite execution time and reducing test suite redundancy. The guidance focuses on a set of best practices that teams can adopt to achieve those goals. The paper reflects on some of the benefits and challenges we faced as part of this case study. It also outlines the tools that were developed to conduct the analysis and support the best practices. We use examples and data taken from the case study to illustrate and emphasize key points.

1. Introduction

For each new release of a major Microsoft product, thousands of new test cases are added to existing test suites. Over time, these tests suites have evolved into huge legacy test beds that consume large amounts of resources for test maintenance, execution and failure analysis.

It appears that a key contributor to our test suite redundancy issue is the turnover and off-shoring of testing staff. The original author of the test code has moved on and the test code may be insufficiently documented or complex. New testers needing to maintain that code are either unfamiliar with it (structure and purpose) or may lack sufficient experience to update it correctly. Instead, they opt to simply copy and slightly modify the test case or write new tests that essentially perform the same purpose as the old test code.

Gaps in the testing process appear to be another contributor. Reviews of test plans and test code are not rigorous enough yet to identify and filter out potential (test case) duplicates in the various stages of test development based on their effectiveness and contributions. For example, reviews do not identify those scenario tests that are effectively subsets of larger end-to-end scenario tests. Rather than deprecating those simpler tests, they remain in the test suite for years. Also, tests are not repeatedly evaluated based on their priority during release or from release to release. Tests that were high priority during the early stages of a development cycle or previous release may now no longer be relevant or as important for product validation.

As a result, test agility is being severely impacted. Test teams that in previous releases were able to run their ‘nightlies’ overnight can now no longer do so, even with extra resources. Instead, teams are finding that their ‘nightlies’ are taking on the order of days to execute (and even longer to analyze). Those teams are seeking ways to retain/regain their test agility with original goals of providing timely data concerning the quality of their builds, e.g. nightly runs – 8 hours, weekly runs – 24 hours, full runs – 48 hours. This problem is further compounded with the rising number of test configurations that require these suites need to be run against – think simply of the combinations of product versions, platforms and SKUs that need to be validated during the product development cycle.

With product functionality changing significantly, there is also the maintenance cost associated with updating the growing number of tests and preventing them from becoming ‘stale’. That cost constitutes an increasing portion of testers’ time. This leads to a situation wherein testers are spending a major portion of their time maintaining and analyzing failures from existing tests, rather than creating new test automation to exercise new product functionality.

Test reliability also appears to be impacted as trade-offs are increasingly being made between writing new test automation and maintaining existing test code. As a result, there are large numbers of test case failures that are being less frequently analyzed by testers. Such situations can in turn have undesirable consequences - true product failures/bugs are

‘lost’ in the thousands of unanalyzed test failures being reported by the test case management/execution system.

The following sections describe the initial phase of a long-term test initiative aimed at regaining our test agility, while maintaining our standards regarding high quality products. In this phase, we aim to characterize and optimize test suites owned by different product units within Developer Division based on two key goals:

- **Test effectiveness** - expressed in terms of broadest possible product coverage and code execution based on the hypothesis that the more code a test case exercises, the more likely it is to find a bug.
- **Test efficiency** - expressed in terms of a reduced number of tests to save on test execution and more importantly, failure analysis time, but also maintain failure detection rates of the original suite.

In Section 2, we describe how we characterized sample test suites and interpreted factors contributing to their test effectiveness and efficiency. Based on that analysis, Section 3 outlines the techniques and tools that were applied to ‘extract value’ out of specific classes of tests, while Section 4 then describes how we ‘squeezed’ the suites to optimize for efficiency. Section 5 highlights some early results by identifying the test case reductions that were achieved, lessons learned and describing key benefits that the teams experienced. Finally, Section 6 outlines the future work that we intend to conduct in subsequent phases of this initiative.

2. Characterizing Test Suites

In a first step, we attempt to characterize various test suites based on their *unique* and *cumulative* contributions to code coverage. Results are aggregated across the product binaries owned by each team. **In this exercise, we use code coverage not so much as an indicator of test adequacy, but focus more on its ability to demonstrate product coverage (or lack thereof).**

The data that we collected is expressed as *block* coverage¹ and is grouped according to test priority. These criteria were selected as the data was readily available at Microsoft - most test teams regularly collect and analyze code coverage data on a per binary basis. They also structure their testing efforts and schedules around running test cases of specific priorities. Thus, mapping coverage data against test priorities helped the dozen teams involved in the study to better understand the results. A typical MS test schedule relies on running P0 or priority 0 tests on a daily basis with P1 tests being run weekly and P2, P3 and P4 tests being run during select test passes or major milestones. P0 tests, often known as *Build Verification Tests*, validate the basic operations of a product, such as

¹ *Block coverage* is a control flow-based coverage criterion that is essentially the same as statement coverage except the unit of code being measured is a sequence of non-branching statements.

installation and start-up. P1 tests concentrate on exercising key customer scenarios and features and so forth.

2.1 Individual Contributions

We assess the individual or independent coverage contributions of each set of tests of a given priority to see, if those tests are able to effectively exercise the product code. Figure 1 compares three Visual Studio product test suites whose tests have been categorized by test priority and charted against % block coverage (y-axis) across product binaries. Column 1 represents P0 tests, column 2 depicts P1 tests and so forth.

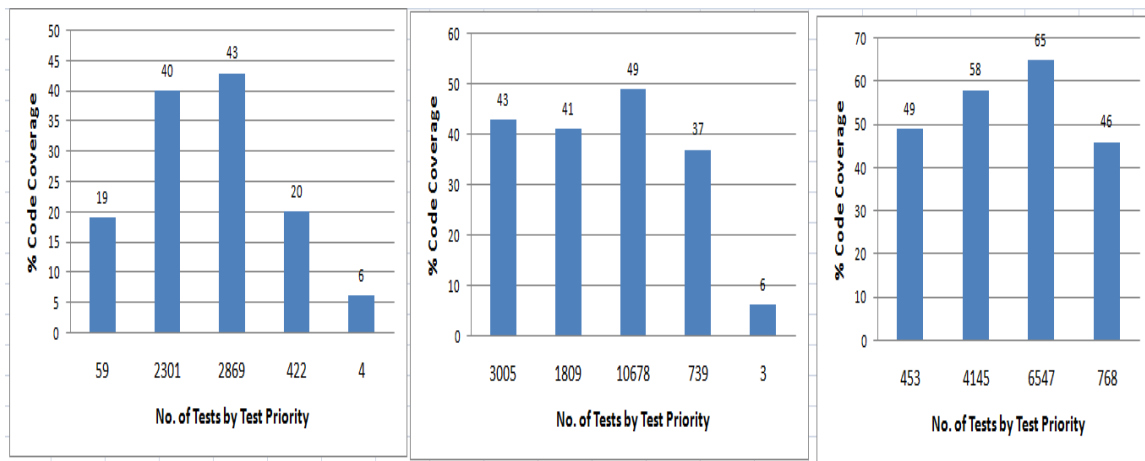


Figure 1: Individual Contributions to Code Coverage by Test Priority

We expect to see (and want to drive towards) an early peak in coverage, so that higher priority P0/P1 tests are executing the major customer scenarios as frequently as possible in a typical test schedule. Subsequent test sets (P2 and beyond) can ideally start to trend downwards in terms of coverage attained, indicating less independent coverage of the product, yet increasing overall cumulative coverage (Figure 2).

We also examined the “yield” per test case ($\% \text{coverage} / \text{total no. of tests}$) to give a *rough* measure of how efficient a particular test category is in achieving its product coverage. Any significant peaks are worth investigating further. By calculating the yield using the data in Figure 1, for example, it became clear that some teams exhibited a wide variation of yields with test suites exhibiting significant peaks with their high (P0) and low priority (P4) tests. The former indicates that some teams have taken time in carefully defining their high priority test sets to broadly exercise the product. The latter typically represent higher priority tests from earlier product releases; these still represent effective regression tests and need to be further examined and leveraged better.

Another interesting point is that the P2 test cases in each product suite appear to provide the best independent coverage compared with the P0 or P1 tests and yet they are only run during major test passes or milestones. This leads us to examine the cumulative coverage of the suites to, for example, see if those same P2 tests also contribute *unique* coverage. In these three cases, they do.

2.2 Cumulative Contributions

Another perspective regarding product coverage is to examine the *cumulative* coverage contributions across those same test suites. We hope to see early peaks in cumulative product coverage using a combination of P0 and P1 tests with coverage trailing off thereafter. This would ensure broadest product coverage during daily and weekly test runs.

The plots in Figure 2 represent cumulative % block coverage against a cumulative count of test cases, e.g. with column 1 depicting P0 tests, column2 showing P0 and P1 tests and so forth. These plots clearly show that the maximum product coverage is only achieved after contributions from P2 tests – the quandary being that these tests are being too infrequently executed. Moreover, P3 and P4 tests provide no increase in cumulative coverage thereafter, yet they represent thousands of additional and *potentially* redundant tests being executed and requiring analysis.

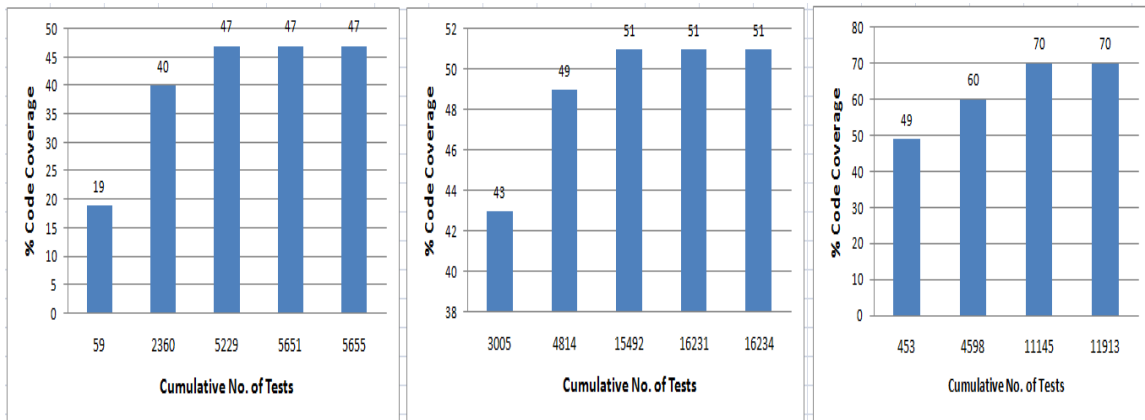


Figure 2: Cumulative Contributions to Code Coverage by Test Priority

3. Addressing Test Effectiveness

In the previous section, we examined different product test suites and attempted to characterize them according to their individual and cumulative coverage contributions. We also reasoned that the P2, P3 or P4 tests provide unique value or test effectiveness to each suite. Thus, in this section, we aim to ‘extract’ and assess that unique value in order to then either upgrade test priority or archive the test case(s).

To achieve this goal in a timely manner, we had to leverage automation that was able to interrogate the code coverage database and determine the differences in traces² between specific sets of tests. Figure 3 illustrates a typical results file showing the sorted, unique coverage contributions of P3 tests in comparison to P0/P1 tests for one of the sample product suites. Tests could be prepared for review based on the ratio of unique blocks

² Traces are the execution traces captured as a result of executing a given test case and expressed in terms of the code blocks traversed.

covered vs. total blocks covered – the closer the ratio is to 1, the more unique the value of that test case. For reference, we have also added the total block count. Full details are given in terms classes/functions covered, so that testers can investigate the code traversed by those tests in depth. At this time, this data is not used for the purposes of collecting metrics or triggering follow-up actions, but could be in future.

Class	Function	Unique Blocks	Total Blocks		Test Case Name
		Covered	Covered	Total Blocks	
Diagram	OnScroll	48	56	79	Branch Root.qa.md.v
VSWCFService	ImportWCFM	35	94	108	Branch Root.qa.md.v
SLClientGener	UpdateErrorLi	33	33	84	Branch Root.qa.md.v
MyApplication	FireChangeNo	28	28	35	Branch Root.qa.md.v
CDaVinciQuer	VSetForClause	26	26	31	Branch Root.qa.md.v
AdoDotNetCon	ToDisplayStrin	26	26	38	Branch Root.qa.md.v
DataTableAnd	OnSplitterMo	23	23	23	Branch Root.qa.md.v
DataConnectic	RemovePrope	22	22	22	Branch Root.qa.md.v
DataObjectIde	FindIdToldMa	21	21	23	Branch Root.qa.md.v
DataConnectic	ModifyProper	20	20	20	Branch Root.qa.md.v
ResourceStrin	WndProc	18	30	31	Branch Root.qa.md.v
Diagram	OnMouseWhe	16	16	25	Branch Root.qa.md.v
ChannelEndpc	GetEndpoints	16	19	23	Branch Root.qa.md.v

Figure 3: Report Indicating Unique Block Coverage by Test Priority (P3 Tests)

4. Addressing Test Efficiency

Now that the unique contributions of a particular test or set of tests of given priority have been analyzed and the appropriate action has been taken, it is time to address our second objective of reducing the overall size of the test suite and thus, achieving better test efficiency.

For this purpose, we again deploy automation that examines the code coverage data and determines a ‘core’ set of tests with the same level of product coverage as the original test set, but significantly smaller in size. As a result, we have a core set of tests that we aim to keep and build upon and a large set of remaining tests that constitute potentially redundant or duplicates (‘dupes’). These need to be reviewed and then either included in the core set or deprecated.

4.1 Selecting a Core Set of Tests

To select this core set of tests, we developed automation that was previously described in [1, 2]. This tool embodies two algorithms - one is capable of determining a truly minimal set of tests using a traditional greedy algorithm [3], the other an algorithm that focuses on maximizing product coverage. In the future, we will be examining the benefits of the first algorithm; this paper focuses on applying the second algorithm. As a result, our core set of tests was slightly larger than the one that would have resulted from applying the

greedy algorithm, but it was therefore also ‘safer’ and had potentially better fault detection capabilities.

Figure 4 illustrates this algorithm at work. For a code coverage matrix that maps tests T1 through T6 against code blocks B1 through B8, the shaded squares depict a test case executing a particular code block. Each of these tests represents a unique customer scenario being independently executed and the corresponding set of code blocks being traversed. For example, scenario test T1 exercises B1, B2, B3 B5, B6 and B8. The algorithm now traverses the matrix, that is, each code block from B1 through B8 and selects the test case with the largest number of blocks traversing not only itself, but also all other blocks in the product binary. By doing so, it is maximizing the product coverage, while also maintaining the original level of product coverage. When the algorithm has finally traversed all blocks, the final set of tests comprises T1, T4 and T6.

Earlier on, we mentioned that this core set was not minimal – in the example below, the application of a greedy algorithm would result in T1 and T2 being picked as the final set. This may not appear to be a big difference in the context of this example, but when test matrices include thousands of tests, this difference is significant. Also, one of the refinements that we are currently implementing is to run this algorithm repeatedly over the previous set of identified ‘dupes’, thereby ordering them and making further review and analysis easier.

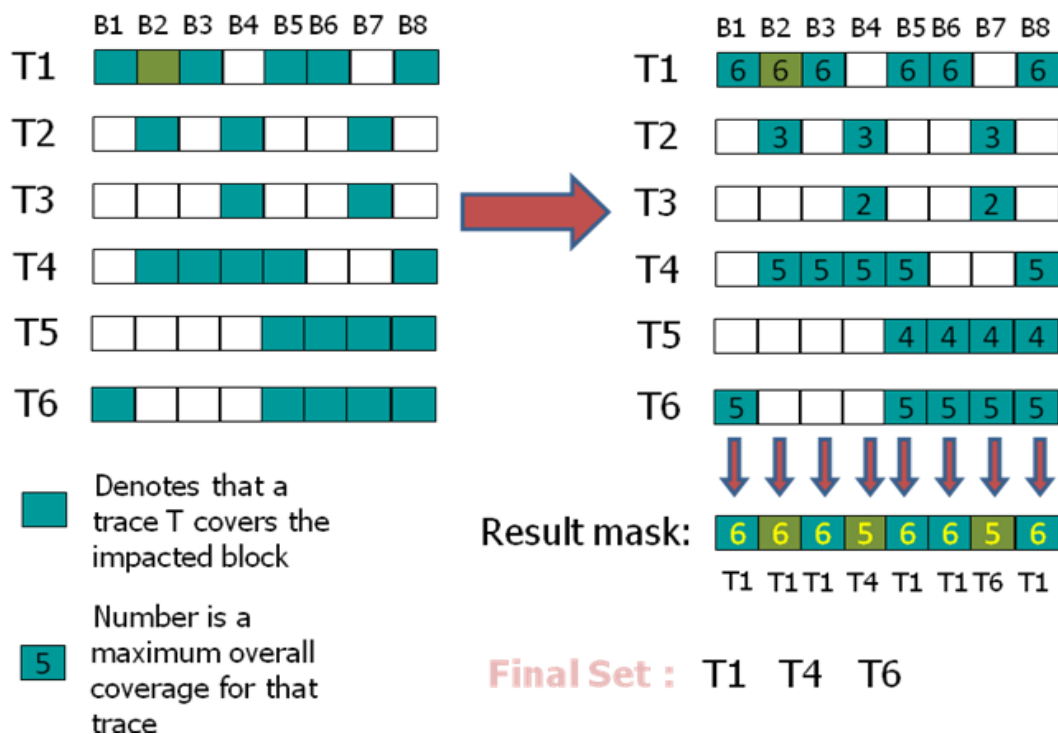


Figure 4: The Maxcoverage Algorithm at Work

It should be noted that before applying the algorithm, we have the ability to filter out any unwanted traces that pertain to failed or unwanted categories of tests, the former often contributing partial product coverage. Furthermore, we explicitly filter out code coverage

contributions from developer unit/API tests as we want to compare and select QA tests, that is, functional tests that validate our customer scenarios. Not doing so, would have led to the algorithm favoring the QA tests and implying that the developer unit/API tests were potentially redundant, which of course they are not.

```
<binary name="microsoft.data.connectionui.dialog.dll" IDComponentInfo="119" traces="249" new_blocks:
  old_blocks="7103" impacted_blocks="7103" deleted_blocks="0" total_blocks="7103">
- <sets>
  - <set number="1">
    - <traces>
      <trace IDTrace="13550" impacted_block_coverage="2455" new_impacted_block_coverage="0"
        old_impacted_block_coverage="2455" blocksaddedtoaset="2455" totalcoverage="2455"
        cumulativecoverage="2455" />
      <trace IDTrace="27209" impacted_block_coverage="2299" new_impacted_block_coverage="0"
        old_impacted_block_coverage="2299" blocksaddedtoaset="482" totalcoverage="2299"
        cumulativecoverage="2937" />
      <trace IDTrace="27211" impacted_block_coverage="2142" new_impacted_block_coverage="0"
        old_impacted_block_coverage="2142" blocksaddedtoaset="268" totalcoverage="2142"
        cumulativecoverage="3205" />
      <trace IDTrace="27148" impacted_block_coverage="2165" new_impacted_block_coverage="0"
        old_impacted_block_coverage="2165" blocksaddedtoaset="147" totalcoverage="2165"
        cumulativecoverage="3352" />
      <trace IDTrace="27213" impacted_block_coverage="2058" new_impacted_block_coverage="0"
        old_impacted_block_coverage="2058" blocksaddedtoaset="43" totalcoverage="2058"
        cumulativecoverage="3395" />
      <trace IDTrace="27408" impacted_block_coverage="2236" new_impacted_block_coverage="0"
        old_impacted_block_coverage="2236" blocksaddedtoaset="32" totalcoverage="2236"
        cumulativecoverage="3427" />
      <trace IDTrace="27214" impacted_block_coverage="2053" new_impacted_block_coverage="0"
        old_impacted_block_coverage="2053" blocksaddedtoaset="30" totalcoverage="2053"
        cumulativecoverage="3457" />
      <trace IDTrace="27153" impacted_block_coverage="2001" new_impacted_block_coverage="0"
        old_impacted_block_coverage="2001" blocksaddedtoaset="18" totalcoverage="2001"
        cumulativecoverage="3475" />
      <trace IDTrace="13548" impacted_block_coverage="2023" new_impacted_block_coverage="0"
        old_impacted_block_coverage="2023" blocksaddedtoaset="10" totalcoverage="2023"
        cumulativecoverage="3485" />
      <trace IDTrace="746220" impacted_block_coverage="1533" new_impacted_block_coverage="0"
        old_impacted_block_coverage="1533" blocksaddedtoaset="3" totalcoverage="1533"
        cumulativecoverage="3488" />
    </traces>
  </set>
</sets>
- <traces_id_name_mapping>
  - <trace IDTrace="13550">
    <![CDATA[ Branch Root.qa.md.wd.Bizapp.AxPro.Data.DataSet.Designer
      [ds].Nightly.dsNightlyTableAdapterESP:1.1.385947.35669.716663.P0.Pass ]]>
```

Figure 5: Sample output from TCIndexer - Test Selection Tool

Figure 5 shows the XML output that is generated by the tool embodying the algorithm. It generates a detailed report for each binary in the product as well as a summary report. Each detailed report indicates the number of code blocks being considered; in this case, 7103 code blocks. Furthermore, it indicates how each test case identified by a *<trace IDTrace/>* attribute contributes to overall product coverage as well as providing a *cumulative coverage* block count. As can be seen in this example, the *blocksaddedtoaset* slowly decreases and the cumulative count rises correspondingly to its final value of 3488 code blocks, which is equal to the original coverage level for that binary. Also, for each test selected, the tool outputs the complete trace string, a piece of meta-data that describes

the test case and which includes information such as test priority, test case name, pass/fail execution status, etc. In Figure 5, the test cases are highlighted within the `<trace_id_name_mapping>` attribute.

With data collected and processed, Figure 6 shows the results of applying the Maxcoverage algorithm to our three sample product suites. The stacked bar chart indicates the number of tests that are part of the core set (Series 1) with potential duplicate tests represented in the upper stack (Series 2). The x-axis depicts the cumulative number of tests in the test suite by test priority, while the table shows data values. So, for example, in the second bar chart, the set of P0/P1 tests contains a total of 4814 tests with 1230 being identified as part of a core set and the remaining 3584 tests being deemed potential duplicates by virtue of not being included in the core set.

The key point of interest is the ratio of core tests vs. the potential dupe sets. For example, there is significant variation in this ratio for the P0 tests, the set of tests that should be run on a nightly basis. In the second chart, the algorithm identified a set of 845 P0 tests and 2160 potential duplicates, indicating that nearly two-thirds of the tests were potentially redundant. Compare this to the first chart, where the algorithm was not able to identify any potential duplicates and thus the core set reflected the original test set! This indicated to us that this set of 59 P0 tests provides broad and unique product coverage with every single test case. In contrast, the other two suites show appreciable potential redundancies and would need to be reviewed.

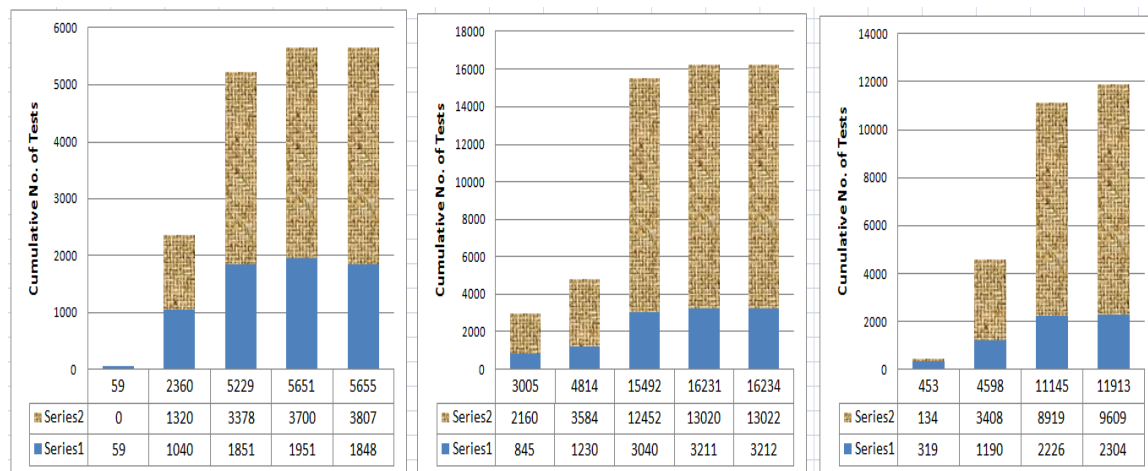


Figure 6: Ratios of Core Set vs. Potential Duplicate Tests by Test Priority

4.2 Reviewing Potential Duplicate Tests

Prior to reviewing and analyzing potential duplicate tests, it is important to realize and understand the limitations of code coverage and how to leverage the generated data discussed above. The code coverage data is being utilized here simply to indicate how each functional test in the suite has traversed the product code. However, it does *not* capture any temporal (timing) or frequency data for that test case. It does not, for example, indicate the order in which a given set of code blocks was executed or the number of times a code loop was taken. We only know that at some point, a specific code

block was traversed by a test. Thus, two trace vectors resulting from the execution of two different tests maybe identical in terms of the code blocks executed, but functionally those two tests may be targeting different customer scenarios or they could be variations of the same or similar customer scenario – we simply cannot tell. Having said that, code coverage data is readily available and the above analysis is ‘cheap’ enough in terms of performance and scalability.

4.2.1 Importance of Stakeholders

A first and crucial step in the review process is to ensure that the results generated above need to be disseminated to key stakeholders, that is, the domain experts within the product test teams. These stakeholders should represent senior members of each test team who are familiar with their feature areas and the tests that traverse the corresponding feature code. They also need to have worked on the team for some time and have knowledge regarding the evolution of their test set. The reason for emphasizing this point is that we found appreciable differences between offshore and in-house staff when reviewing and analyzing the potential duplicate sets and comparing them to core sets. The experienced (and thus more confident) senior testers were able to identify a larger number of *actual* duplicate tests. In contrast, offshore testers were less familiar with the tests (and thus more cautious) at deciding whether a potentially dupe was *actually* one. Thus, the number of actual dupes identified by those offshore teams can be significantly less.

4.2.2 Sample Recommendations

Once the stakeholders know the tests contained in the core set, their task of comparing these to the set of potential dupes for their feature areas begins. As this is ongoing test initiative, we are continuously evolving and refining the recommendations for other teams to leverage. Here are some samples of the prescriptive guidance provided:

- **Sub-scenario Tests** – many potential dupes exercise sub-scenarios of larger scenario tests. Our recommendation is to archive these, whenever the core set already contains a larger scenario test that ‘covers’ the sub-scenario or if appropriate, merge a number of sub-scenario tests into a new scenario test that can be added to the core set.
- **Regression Bug Finders** – if the potential dupe set contains tests that have found important regression test bugs, it would be wise to keep these and add them to the core set.
- **Tests with Extensive Verifications** – when considering test code, many potential dupe tests differ from their core set counterparts or other dupes by the amount of verification code that has been added at key points. Our recommendation has been to either keep all of these tests or merge them into a smaller set of tests.
- **Boundary Tests** – it is very unlikely that code coverage could distinguish these types of tests unless a significantly different execution path has been taken. Thus, our recommendation is to keep these tests.

- **Data-driven Tests** – some potential dupes represent scenarios in which Visual Studio code, for example, interfaces to SQL. Code coverage cannot reflect this interaction via traces and thus all of these tests must be retained.
- **Scenario Ordering** – as code coverage data cannot identify code execution order, scenario tests that have exercised the same code blocks, but in a different order cannot be distinguished. Thus, these potential dupes need to be examined more closely. As a result, the dupe is either archived, if its intent is to validate the equivalent customer workflow/functionality or added to the core set, if the intent is distinct from the test(s) in the core set.

5. Preliminary Results and Analysis

5.1 Goals

Before sharing some of our *preliminary* results from this ongoing study, we wanted to state its goals:

- Examine the accuracy of the Maxcoverage algorithm at picking a core set of tests
- Identify the ratio of potential vs. actual redundant tests (as determined by testers)
- Compare the fault detection capabilities of the final test set vs. the original set³

At the time of writing, about a dozen teams from across Developer Division are actively participating in the case study and working on various stages of analyses. These include test data preparation, test suite characterization, potential duplicate test identification and potential dupe review and deprecation.

The teams can best be characterized as representing products ‘up and down the Visual Studio stack’ from the Visual Studio platform to Visual Studio Team System. The teams own code and test suites that vary in terms of size and age. Focusing on the test aspect - some teams own only a few hundred tests that have been developed over the past one or two product release cycles; others own suites that have been around for ten years or more and include thousands of tests.

5.2 Results from Participating Teams

For this paper, we wanted to share the early results from two product teams that have completed one of the later phases, the potential dupe review phase. Neither team has undergone any kind of optimization of their test suite before. The first team maintains a true legacy test suite of nearly fifteen thousand tests, which have evolved over ten years

³ Unfortunately, at this time, we are unable to report on the fault detection capability of the final set of tests as none of the teams has had sufficient time to complete the potential dupe review and then compare and draw conclusions concerning the fault detection capabilities of the original and final set of tests.

or more. The second team maintains a more recent suite of tests numbering several hundred tests.

The first team decided to analyze their test suite based on key features, so the results in Figure 7 represent the results for one of those key features, rather than for the entire product⁴. If we examine the data in Figure 7, we see that testers had developed approximately 1600 tests of varying test priority for this feature. The domain expert was then presented with a set of core tests (746 tests) and potential dupes (883 tests) as determined by the Maxcoverage algorithm. He then examined and compared each test case from the core set with each of the potentially redundant tests. This comparison was conducted at a rate of about 5 minutes per test comparison.

Of the 883 potential dupes, 715 tests were actually found to be redundant and could initially be disabled in the test case management system (and later on deprecated). This indicates that there was only a 10% variation (accuracy) in the number of core tests originally calculated by the algorithm and the final core set determined after the manual review phase.

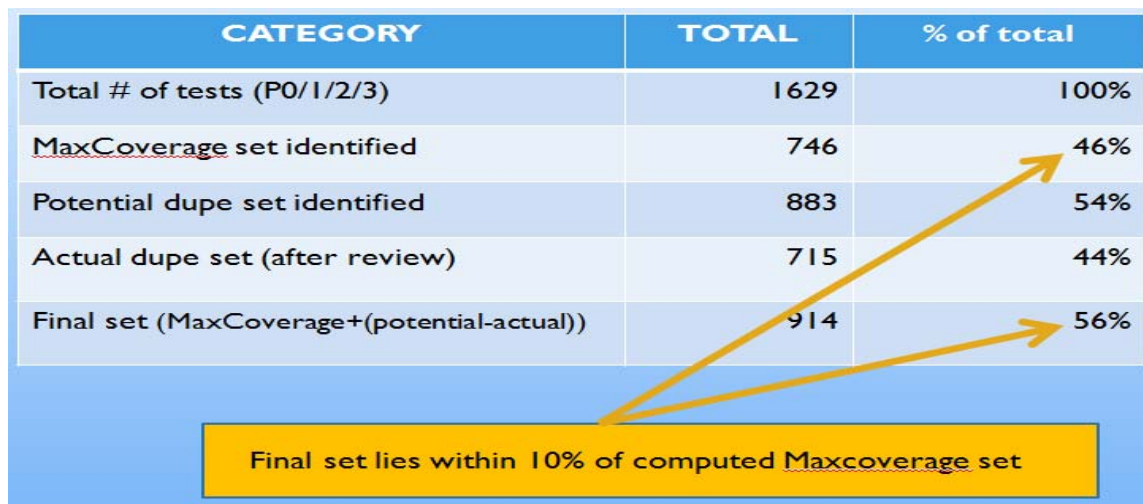


Figure 7: Results from Team #1

For the second team, we can share the results for the entire suite as their test cases numbered in the hundreds and their domain expert had knowledge of their entire suite. Their original test suite comprised of over 650 tests with the Maxcoverage algorithm identifying approximately 200 tests as the good core set and the remainder (over 450 tests) as potentially redundant. After analysis, they too found that the algorithm had identified the core test set to within 10% accuracy. So, the final test set was just over 200 tests.

⁴ Test suite characterization results for this team and their entire suite of tests can be found in the chart 3 of Figures 1, 2 and 6.

5.3 Benefits and Lessons Learned

In this section, we wanted to highlight key lessons learned so far as well as discuss benefits. While both teams agreed that performing such an exercise was time-consuming from the point of view of analyzing the potential dupes manually, the reductions being achieved in both cases, more than offset that cost.

5.3.1 Test Execution (and Failure Analysis) Time

A good example for improving test agility comes from the second team described earlier. Its suite of more than 650 automated UI tests required more than 2 days to run on a single machine with several team members needing to investigate failures. Their tests were unstable and frequently failed. After completing the above exercise, the team then focused on improving test reliability. As a result, this smaller set of tests can now be run overnight as part of a nightly test run with tests passing at a much higher rate and only one tester required for analyzing failures. Product (code) coverage has been verified as being equivalent of the original test suite. By focusing on this new core set of tests and test failures from these tests, the team (re)discovered three high-priority (P0/1) bugs in subsequent test runs. Previously, they had been swamped with test failures and had missed them.

5.3.2 Test Planning and Review

Participating in this exercise, the first team realized that they required improvements to their test case planning and test code review processes to prevent the most common mistakes being made by testers in future. One example that is driving the need for more careful code reviews is the existence of large numbers of sub-scenario tests that had not frequently enough been reviewed and merged into larger scenario tests. Another example highlighted a loophole in their test tooling/processes that resulted in testers being able to create tests that contained setup/teardown code, but no actual test case body. As a result, these ‘junk’ tests were not exercising the product, but were always passing! It turns out that appreciable numbers of such tests were wasting test resources. The exercise also helped in reviewing and prioritizing the remaining scenario tests in terms of importance and risk.

5.3.3 Test Code Knowledge and Regular Checkups

A key lesson learned by both teams has been that product coverage and value added by new (and existing) need to be assessed on a regular basis in order to maintain test effectiveness and efficiency. This also benefits team members’ knowledge of the test and product code, which given the staff turnover is very valuable. As part of this study, we are recommending that teams consider regular checkups to maintain their test agility – prior to test passes (in the test preparation phase), at major milestones (MQ or milestone quality) or on a more frequent, *per tester* basis. The latter would be the most effective model whereby testers themselves leverage the tools described in this paper and to ensure efficiency and effectiveness of their own tests.

6. Conclusions and Future Work

In this paper, we have examined techniques and tools that leverage code coverage data to help test teams focus on their test suite effectiveness and efficiency. We highlighted the application of techniques and tools to improve their test agility and the steps that need to be taken to achieve that goal. Furthermore, we illustrated this process by means of an ongoing pilot involving test teams from the Developer Division and reported preliminary results. Finally, we reflected on key improvements that teams are making as a result of participating in this exercise.

It is also important to note that at this time, the review and analysis process is completely *manual* in nature. Having said that, we are seeking and evaluating new tools that promise us better duplicate test identification as well as reduced review and analysis times. Currently, we are exploring the Spirit tool [4] that not only compares code coverage traces and analyze their differences and similarities, but also leverages data from execution profiling and static analysis techniques applied to the product binaries to accelerate this otherwise arduous manual task. We will be reporting the results and impact of this tool on our initiative in a future conference paper.

7. Acknowledgements

I would like to thank Larry Sullivan, Director in Developer Division, for his ongoing support. I also want to express my gratitude to my Microsoft colleagues, particularly Carey Brown, Xiang Zeng, Shaun Phillips, Mike Taute, Alex Teterev and Jacek Czerwonka for their contributions, support and discussions concerning this work as well as the Developer Division code coverage champions for helping me collect the necessary data. Thanks also go to my reviewers whose invaluable feedback and comments are greatly appreciated.

8. References

1. J. Hartmann, “*Applying Selective Revalidation Techniques at Microsoft*”, PNSQC, pp. 255-65, Oct. 2007.
2. A. Srivastava and J. Thiagarajan, “*Effectively Prioritizing Test in Development Environment*”, International Symposium in Software Testing and Analysis (ISSTA), pp. 97-106, 2002.
3. V. Chvatal, “*A Greedy Heuristic for the Set-Covering Problem*”, Math. Operations Research, pp.233-5, Aug. 1979.
4. V. Vangala, J. Czerwonka and P. Thalluri, “*Test Case Comparison and Clustering using Program Profiles and Static Execution*”, European Software Engineering Conference and ACM Foundations on Software Engineering, Aug. 2009.

Where are You in Usability?

Kelcie L. Anderson

kelcie.anderson@live.com

Abstract

As a QA professional, how are you contributing to the usability of your product? The usability, or user experience, of a product has steadily gained in importance over the last decade. Once, only a few people knew the phrase, but now the term usability is readily bandied about.

With a solid understanding of a good UI design process, you can effectively collaborate with the UI team throughout the development life cycle. In addition, you are in the unique position to overcome the following barriers to usability: (1) Production code is not implemented per the UI design, (2) the UI design cannot be implemented as specified, causing the developer to change the design “on the fly,” or (3) the company is not operating with the above UI model.

Biography

Kelcie Anderson is a Project Management Professional with a background in program management and usability engineering management. She spent 9 years in the User Experience group at Tektronix. She is currently streamlining the product development process at Acumed.

We've all experienced firsthand products with poor usability. Like the web application that displays a 16 digit code and then two clicks later asks you to enter the full number. What, were you expected to memorize it? Or the new software upgrade that crashes when you try to open a file created with the previous software version. Every time you feel the pain of a usability issue in your own life, you vow to do everything you can to ensure your own company's products are user friendly.

Usability and user interface (UI) design have gained importance to both customers and companies in the last decade. Fierce competition has created an environment where it is no longer good enough for a product to just work. Winning products work well for the customer, supporting their actual tasks and helping them accomplish their goals. This trend is good news for quality analysis (QA) professionals because good QA experts are characterized by sensitivity to the customer. The goal of QA is to ensure the customer receives a quality product that works well for them.

QA is well positioned to catch usability errors before the product is released. But it is difficult to test your way to a good user interface design. QA professionals can play a more influential role in the usability of the product by influencing the design throughout the design cycle. To increase your contribution to usability, you must understand the process for creating a good user interface design, how QA can contribute to the process, and how to recognize barriers to UI implementation that you are uniquely qualified to overcome.

Process for User Interface Design

Good user interface design is dependent on three key elements: customer-based requirements, iterative design, and iterative testing. To start, the UI team needs to understand the real customer requirements. These requirements are far more than a list of features. Customer-based requirements require a deep understanding of the user tasks and goals that the product is meant to support. The UI team puts themselves in the customers' shoes to understand the motivations, pressures, workflow, and pain points that customers experience. This in-depth knowledge provides the basis of customer problems the product can solve. Then, and only then, the customer problems are converted into a set of technical requirements.

Take, for example, a program I worked on at Tektronix. My team was in charge of defining and designing a new voltage probe to work with the new generation of oscilloscopes. The probe was capable of several technical features, including bandwidth limiting for increased resolution. Instead of designing a button or user interface to support the probe bandwidth limits, the UI team first conducted interviews with real customers. The team quickly realized that the customer didn't require a probe bandwidth limit feature. The customer was looking for options to limit the bandwidth of the system—both probe and oscilloscope together. By fully understanding the customer requirement, the UI team shifted to thinking about bandwidth limiting as a system instead of a specific probe feature.

With a solid set of requirements that are based in real customer needs, the UI team then proceeds with the design. The first design step is to produce use cases and usability goals—concrete tools

that narrate the customer task and guide the design. With the vivid, realistic detail of the requirements and use cases at hand, the user interface takes shape. But UI design, just like most good design, is an iterative process. The UI team does rapid design, test, and design modification to complete the full user interface.

The testing tool of choice for rapid UI design is usability testing. This form of testing involves recruiting real end customers to sit in front of a design prototype and complete various realistic tasks while being observed by the UI team. This feedback is immediately incorporated into the UI design.

Because UI design requires many different skill sets, the team is typically comprised of several individuals with different roles. A good UI design team includes a usability engineer, marketing representative, software architect (or developer), and graphic designer. These individuals bring the following necessary skills to the UI design:

- Usability engineer: customer interviewing and needs assessment, UI design, usability testing
- Marketing representative: customer interviewing, market size analysis, customer recruiting
- Software architect: technical feasibility, technical innovations
- Graphic designer: UI design

The UI design team typically finishes the design and turns it over to a software team for implementation and to QA professionals for production code test. You have undoubtedly noticed that QA is not included in the UI design team. But QA professionals have a crucial role to play in the successful implementation of a usable product.

QA Role in Usability

You may already be using a tool for production testing that is invaluable at the beginning of the design process: workflow testing¹. Workflow testing relies on solid task analysis and realistic use cases. The UI design team will be creating these tools as part of the design process. As the QA expert, you can coordinate with the UI design team and utilize the same workflow they used for your own testing. Better yet, you can contribute your expertise to create, review, and revise workflow scenarios with the UI design team at the beginning of the process.

An example of this beneficial collaboration between the UI team and QA early in the design phase comes from a colleague at General Electric. The UI team was creating a new wizard to aid doctors in filling out forms. No one knew the ins and outs of the existing forms better than the QA professional. The UI team asked for a review of their use cases and design requirements. The QA professional spotted a big hole: the UI team had not included a use case for doctors who want to skip around to various sections of the form rather than fill it out in a linear fashion. Through her experience with workflow testing the existing forms, she knew the “map view”

¹ Testing for the User Experience *User Workflow Testing*, By Lanette Creamer, October 2008 from http://www.pnsqc.org/wp-content/uploads/2008/10/creamerc_pnsqc2008.pdf

feature was designed to aid doctors in this very scenario. The UI team had been unaware of the existing map view feature and the requirement to skip around a form.

Collaboration with the UI design team can continue through the iterative design and testing phase. For usability testing, you can once again offer your expertise in creating, revising, and reviewing the testing. When testing begins, you can observe several of the usability tests to further deepen your understanding of customers and their goals. You may continue working with the UI design team during your testing of the production code, using members of the UI design team as expert testers. And better yet, you could work with the team to recruit real end customers to run through your workflow tests.

Overcoming Barriers to Usability

Of course, the description of the UI design process includes the implicit assumption that the design will be implemented per specification and the end result will be a highly usable product. In the real world, this assumption is often flawed and QA can be there to bring the usability of a product back on track. The following barriers to usability can arise: (1) Production code is not implemented per the UI design, (2) the UI design cannot be implemented as specified, causing the developer to change the design “on the fly,” or (3) the company is not operating with the above UI model.

The first barrier to completing a successful UI design is the case where developers are not coding to the design specification. This often occurs when a company has an established software requirement control system and is just starting a cross-functional UI design model. Sometimes, the UI team is not fully versed in the established procedures for how the development team tracks and tests against requirements. They may write a nice user interface specification document without fully integrating it into the technical specification documents that the developers operate against.

For example, at Tektronix the System Requirement Specification (SRS) was an established document the SW group had been using for many years. The User Interface Specification was a subset document that contained the technical parameters for each element of the UI. When a UI Team was formed, they began writing the User Interface Specification (UIS), with more emphasis on the task flow and how the user navigated from one element to the next. However, the software team was accustomed to coding based on the SRS document and largely ignored the UIS. After the coding was complete, they would update the UIS with the parameters and screen shots of what they had decided to implement.

In this circumstance, you, as the QA professional, are well versed in the procedures for documenting software requirements since these are the documents you test against. To increase the usability of the product, you can work with both the development team and the UI design team to gather all relevant specifications and ensure document control procedures are followed. You are then in the invaluable position of being able to highlight any discrepancies between the different documents. You can educate both the development team on the existence of the UI design document and the UI team on the existence of the software requirements document. Because both you and the design team are focused on ensuring an optimal customer experience,

you should take this as an opportunity to coach them on the documentation procedures at your company.

The second barrier—the UI design cannot be implemented as specified—occurs quite often. No UI design specification perfectly anticipates the difficulties that may arise when implementing the design. The software developer must adjust the design but might not consult with the UI design team on the changes. Without the big picture of the customer problem and how the design was intended to solve the problem, these changes may reduce usability. Sometimes the developer feels forced to make the change without consulting the UI team due to tight deadlines. Occasionally the UI team has been disbanded or assigned to another product and is not readily available for a consult. In either case, the integrity of the design has been compromised and must be evaluated for the impact on usability.

As the QA professional, you are again in a good position to notice discrepancies between what was specified and what was implemented. In this circumstance, you can seek out the UI design team to highlight the differences and ask for a review of the new design. Because you collaborated with the UI design team earlier in the design process, you should be able to get their attention now, even if they have moved on to other products. In fact, most members of the UI design team will be happy to know that you're on the job, ensuring the usability of the product is preserved.

But what about those companies that do not use the UI design process described in this paper? What if the UI design is done by the developer who is writing the code—alone and at his desk? In these circumstances, QA has an even greater responsibility to start early in the product design phase to ensure usability. You should start by asking the following questions:

- Who is responsible for the design?
- Have any use cases been created or task analysis been done?
- Who reviews the design for usability?

The answers to these questions will guide your actions. The most important thing you can do is to find the use cases or, if none are available, create the workflow test. Seek out other customer-minded representatives, such as marketing, and ask for a review of the workflow you have created to further incorporate customer perspectives from different roles within the organization. Share this information with the person responsible for the design and let them know you will be testing against these criteria. Through your actions, the developer has an opportunity to design to real tasks instead of designing in isolation.

A good QA professional is focused on delivering a product to the customer that not only works but works well to support their tasks. By becoming more involved in the user interface design process throughout the requirements, iterative design, and iterative testing phases, you can enhance the quality, user experience and success of your company's products.

An Empirical Study of Data Mining Code Defect Patterns in Large Software Repositories

Kingsum Chow, Xuezhi Xing, Zhongming Wu and Zhidong Yu
Software and Services Group, Intel Corporation
[kingsum.chow, xuezhi.xing, zhongming.wu, zhidong.yu}@intel.com](mailto:{kingsum.chow, xuezhi.xing, zhongming.wu, zhidong.yu}@intel.com)

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 30 technical papers and patents.

Xuezhi Xing is a software engineer intern. His research work includes software maintenance, software quality and software engineering paradigms. He is pursuing the Ph.D. degree in University of Science and Technology of China.

Zhongming Wu is a software engineer intern. He is pursuing a MS degree in Computer Science and Engineering from Shanghai Jiao Tong University. He is interested in information security and related research.

Zhidong Yu is a software engineer in the Intel SSG. Zhidong has been working on performance analysis of enterprise Java applications and server virtualization software. He received his MS degree from Shanghai Jiao Tong University in 2001.

Abstract

There has been a growing interest in mining software code defect patterns and using this knowledge to identify potential problems [2, 8-15]. To understand the benefits of such methods, we applied them to several large software repositories. We learned the effectiveness and the limitations of applying these methods. These methods are called “static analysis” as they analyze the code for defects without execution. They are different from the traditional static analysis as they apply data mining in the analysis, while the traditional static analysis uses program analysis, such as data flow analysis and control flow analysis. There is a trend to combine the data mining methods and program analysis techniques to gain more effective results.

Many tools are available to detect software defects. But if the tools have no knowledge about what to check they can't find defects [5]. There are common defect patterns such as buffer overflow, null pointer dereference and several tools address these patterns such as Purify [19], FindBugs [20]. However, application specific defects can be difficult to find, probably because there are few common patterns among different applications. Therefore some approaches, such as DynaMine [2], try to explore the patterns of specific applications automatically, while others try to provide description based methods to describe these patterns as assertion statements. Examples of tools and methods that fall into this later category are contract programming, AOP (Aspect Oriented Programming), PQL [3], and Metal [4]. The automatic approaches such as DynaMine may have difficulty in offering a good solution to explore complex patterns. The description based approaches such as PQL are often powerful at describing patterns, but they require manually constructing the specifications. Such tasks can be overwhelming [5].

In this paper, we dive into industry-level projects such as Harmony [16] an open source virtual machine to analyze and summarize their defect patterns using different pattern analysis methods. We gain some insights into the classifications of common code defect patterns. We make use of data mining methods to extract usage patterns automatically from the source code of different projects. We also look into the issues tracking system and revision history to analyze and summarize patterns. We will apply these tools and methods to detect pattern-related defects in new source code as well. Several methods are evaluated for their effectiveness to detect defect patterns.

The contributions of this paper are: (1) an empirical study to evaluate the effectiveness of data mining software code defects using a few large software repositories, and (2) insights into the characteristics of the software systems and the common code defect patterns.

1 Introduction

Defect patterns are important for defect-finding tools. Without such patterns, defect-finding tools cannot find bugs [5]. The use of such tools is valuable for initial software development and maintenance. Recently much attention has been paid to find defect patterns, but defect patterns may be unique and complex. Often only the programmer, who has the knowledge and experience with the code, can summarize the defect patterns. This process is often costly and not scalable. Software code repositories can often reflect the programmer's knowledge and experience. Some researchers try to mine this knowledge or experience to aid software development and maintenance using data mining methods. Research in identification of defect patterns has emerged [2, 8, 9, 10, 11, 12, 13 14]. Still, there is a gap between the capability of data mining methods and the complexity of defect patterns. There are two questions: (1) how large is the gap between them, (2) what kinds of defect patterns can current data mining methods deal with.

To answer both questions, we downloaded 51 Java projects from Apache [22] to study the defect patterns. We picked one of the Apache projects, Harmony [16], to get some insight into the characteristics of software defect patterns. The aim of Harmony, an open source project, is to produce an independent and compatible implementation of the Java Standard Edition 5 JDK under the Apache License. It also includes a community-developed modular runtime architecture. It has more than 1.25 million lines of code. Using Harmony as a case study has an added advantage in that it contains both C/C++ and Java code for us to study. We classified code defects and evaluated some data mining methods, such as Apriori [6], that are used to detect them.

While we provide a detailed defect classification for the Harmony project, the manual classification process was not repeated for the other projects. Instead, we developed a tool set to apply data mining methods to all the Java projects. By combining both stages of our work, we found the following:

1. Project configuration and condition checking related defects constitute about half of all the defects;
2. Current data mining methods may only detect a small portion of the defects
3. More complicated defect patterns can be detected by combining data mining methods and program analysis techniques.

2 Software Defect Classification and Analysis

We manually analyzed the code defects for the Apache Harmony project. We found both the orthogonal defect classification [25] and the root cause defect analysis [26] helpful. However, we could not employ these methods directly because we did not have access to the entire development process. In our manual analysis, we found the following defect classes.

- Requirements Engineering (RM)
- Value Overflow (VO)
- Project Configuration (PC)
- Conditions Checking (CC)
 - Neglected Conditions (NC)
- Function Calls
 - Deprecated Functions (DF)
 - Missed Functions (MF)
 - ◆ Function Call Pairs (CP)
 - ◆ Function Call Sequence (CSq)
 - ◆ Function Call Structures (CSt)
 - Extra Functions (EF)
- Data Usage
 - Value block (VB)
 - Value concurrency (VC)
 - Value error(VE)

2.1 Requirements Engineering (RM)

Some defects are caused by misunderstanding of requirements, or a change of requirements. These kinds of defects by nature cannot be easily avoided by code-based approaches alone.

2.2 Value Overflow (VO)

A value overflow includes integer overflow and buffer overflow. Insufficient checking of this may cause bugs. It may be impractical to check every possibility of a value overflow because of the potential performance impact. An example containing the problem and its fix (e.g., Harmony issue #6204 [16], is given in Figure 1:

```
private int computeElementArraySize() {  
-   return (int) (((long) threshold * 10000) / loadFactor) * 2;  
+   int arraySize = (int) (((long) threshold * 10000) / loadFactor) * 2;  
+   return arraySize < 0 ? -arraySize : arraySize;  
}
```

Figure 1 A value overflow defect example: Harmony Issue 6204

The symbol “-“ means a deleted line in patch and “+” means an added line in patch. Examples in the rest of this paper follow this notation. The above cast from `long` to `int` type in deleted line leads to overflow and a negative `arraySize` if threshold is too big. In Harmony, if `NegativeArraySizeException` occurs, it is often the result of this kind of defect. Just this defect class results in 15 defects in the Harmony project.

2.3 Project Configuration (PC)

Project management defects include files or libraries that are not synchronized with respect to a change. This kind of defect often involves non-source code files; such as ant build files, libs, jars, and dlls. For example in Harmony-795 [13], the build process is broken because of the omission of “`serializer.jar`” in the ant class path. In the Harmony VM module, this kind of defect accounts for 15% (3/19) of all bugs at the time of writing this paper.

2.4 Condition Checking (CC)

A condition checking defect occurs when some needed condition is not checked correctly. It can be described like this: “Always check conditions in set C before/after A”. A common example is the need to check the validity of a pointer before its use. An example containing the problem and its fix is given in Figure 2.

```
static void wait_finalization_end(void) {  
    hymutex_lock(&fin_thread_info->end_mutex);  
-   while(unsigned int fin_obj_num = vm_get_finalizable_objects_quantity()) {  
+   unsigned int fin_obj_num = vm_get_finalizable_objects_quantity();  
+   while(fin_thread_info->working_thread_num || fin_obj_num) {  
+       .....  
+       fin_obj_num = vm_get_finalizable_objects_quantity();  
+   }  
    hymutex_unlock(&fin_thread_info->end_mutex);  
}
```

Figure 2 A condition checking defect example: Harmony Issue 3671

In Figure 2, before the wait action is performed, the conditions

`(fin_thread_info->working_thread_num || fin_obj_num)`

must be checked.

Neglected Conditions (NC) [8, 9] can be regarded as a subclass of this pattern. Neglected conditions refer to (1) missing conditions that check the receiver or arguments of an API call before the API call or (2) missing conditions that check the return values or receiver of an API call after the API call [8]. A recent

study conducted by Chang et al. [9] shows that 66% (109/167) of defect fixes applied in the Mozilla Firefox project are due to neglected conditions.

2.5 Function Call Usage

Function calls are basic elements in writing programs. There are often some specific patterns of function calls. If such patterns are violated, it often results in defect. Several examples are given below.

2.5.1 Deprecated Functions (DF)

Some deprecated functions may cause defects. The designer wants to prevent this kind of function from being called. There are two patterns of this kind, 1) In situation S, call A rather than B; 2) always call A rather than B. Chow and Notkin [1] describe a semi-automatic approach to handling deprecated function calls effectively in the client programs if the library maintainer can specify the formal language proposed in that paper. In the defect fix patches, one-line changes with a function call often relate to this kind of pattern. An example is given in Figure 3.

```
Finref_Metadata *metadata = gc->finref_metadata;  
- metadata->finalizable_obj_set = pool_get_entry(metadata->free_pool);  
+ metadata->finalizable_obj_set = finref_get_free_block(gc);
```

Figure 3 A deprecated function defect example: Harmony Issue 4278

The function `pool_get_entry` is a stack-pop-like function. When this function is applied to the `free_pool` field of `Finref_Metadata` data, it should be replaced by the function `finref_get_free_block(gc)`. Besides getting an entry from

```
gc->finref_metadata->free_pool,
```

the new function does some extra work. This is a very common situation; at a function call, we find that the old function does not deal with some functionality needed by our program, and so we have to wrap or replace the old function call with a new function call. This produces the program pattern: when needing to call an old (deprecated) function, call the new function instead.

2.5.2 Missed Function Calls (MF)

In some specified situations, like handling an error, specific functions must be called. The pattern can be described like “In situation S, always call A”. An example is given in Figure 4.

```
if (bytesRead == 0){  
+   portLibrary->error_set_last_error (portLibrary, 0, 0);  
   return -1;  
}
```

Figure 4 A missed function call defect example: Harmony Issue 4017

The case “`bytesRead == 0`” is the error scenario; using the error process function is missed in the older version.

The recognition of a specified situation is the key step in characterizing this kind of defect, like above defect Harmony 4071. In the current literature, most of the data mining methods focus on the recognition of specified situations which involve function calls. For example: function call pairs, call sequence, and call structures are covered in the following sections.

2.5.2.1 Function Call Pairs (CP)

Some function pairs, like `malloc` and `free`, need to be called together. Also some application-specific function pairs may exist, e.g., `addListener` and `removeListener` in DynaMine [2]. There may be defects if these pairs are not called together.

2.5.2.2 Function Call Sequence (CSq)

Some sequences of function calls need to be called in order. Some researchers [11, 12] have focused on mining such sequences. MAPO [11] mines the API call sequence pattern using BIDE [24] algorithms from open source projects. Mithun Acharya [12] uses partial order relation to mine the API sequence pattern.

2.5.2.3 Function Call Struc (CSt)

There are also more complicated patterns having to do with function calls and some structures are needed to address this kind of pattern. For example Huzefa Kagdi, et al. [13] take into account program's control constructs. They encode function calls with condition statement label. For example, if function **D** is called in an "if" condition statements as `if(D());` this call is encoded by `<if><condition>D</condition></if>`. If function **A** is called inside the `if(D())` block, it's encoded by `<if_cond="D">A</if_cond="D">`. They developed *spminer* [13] to mine the frequent sequence pattern from the encoded function calls.

2.5.3 Extra Functions (EF)

Similar to missed function calls, some functions must not be called in some specific situations. It can be described like "In situation S, don't call A". An example is given in Figure 5.

```
public XMLDecoder(InputStream inputStream, Object owner, ExceptionListener
listener, ClassLoader cl) {
    .....
    - try {
    -     SAXParserFactory.newInstance().newSAXParser().parse(inputStream,
        new SAXHandler());
    - } catch (Exception e) { this.listener.exceptionThrown(e); }
}
```

Figure 5 An extra function defect example: Harmony Issue 6015

The `parse` function call in Figure 5 is an extra function. The fix in `XMLDecoder`'s constructor is to remove it.

2.6 Data Usage

There are often some specific constraints between data, especially data members of the same object in Object Oriented programming languages. Such data needs to be modified in a specified pattern to uphold the constraints. If constraints are not implemented, defects often show up. An example is given in Figure 6.

2.6.1 Value Block (VB)

Some data values are closely related to each other. We call these data a value block. If one datum is not initialized or set correctly, there may be some errors. It is best illustrated by an example in Figure 6.

```
public int read() throws IOException {
    synchronized (lock) {
        .....
        if (pos < count || fillbuf() != -1)
            return buf[pos++];
    + markpos = -1;
    + return -1;
    }
}
```

Figure 6 A value block defect example: Harmony Issue 6110

From the patch in Figure 6, we can see that the root cause of this defect is forgetting to set data **markpos** to -1. The data **pos** and **markpos** are two data members of the same class. If **pos** is greater or equal to **count**, **markpos** needs to be -1. The constraints on **pos** and the value returned by **fillbuf()** ensure that **markpos** is set correctly.

2.6.2 Value Concurrency (VC)

Unaware concurrency may cause related values to be inconsistent. Two kinds of this defect are unexpected concurrency on non-shared values and missing concurrency protection on shared values. An example is given in Figure 7.

```
- private static String name = null;
+ private String name = null;
```

Figure 7 A value concurrency defect example: Harmony Issue 5978

The value **name** is changed to be an instance attribute of a class. By doing this, unexpected concurrency on **name** can be avoided.

2.6.3 Value Error (VE)

An incorrect value causes a wrong behavior in the program. An example is given in Figure 8

```
- factory.createURLStreamHandler(protocol));
+ factory.createURLStreamHandler("jar");
```

Figure 8 A value error defect example: Harmony Issue 6059

The value of parameter should be **jar**.

2.7 Analysis

We analyzed all the 89 closed Harmony defects that were documented between May 1, 2008 and May 1, 2009. The distribution and types of defects are presented in Figure 9.

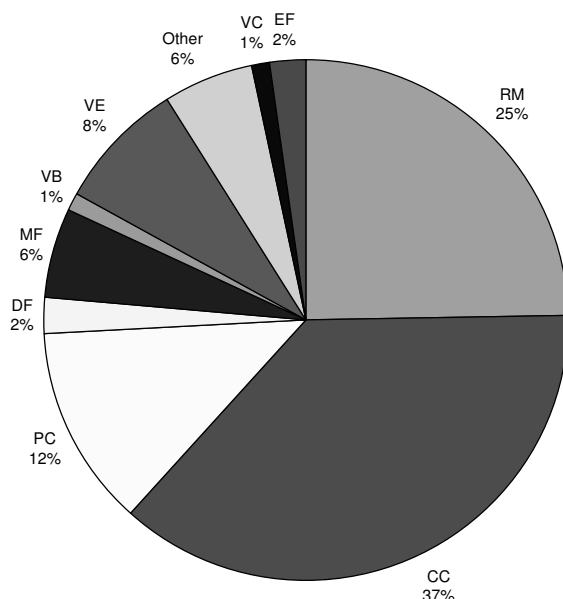


Figure 9 Distribution of defects in Harmony from May 2008 to May 2009.

From Figure 9, we can see that the condition checking defects are most common. The project configuration related defects also have a serious impact, as they constitute 12% of all the defects. Project configuration and condition checking related defects make up nearly half of the total. Note that 25% of the defects are requirement defects and are beyond the scope of this paper.

90% of the defects are classified as being a major loss of functionality in the issue tracking system. All 89 defects cost 1947 days to fix. In terms of the total number of days to fix defects, the condition checking and project configuration defects constitute 38% and 14% respectively, roughly in proportion to the distribution of defects.

3 Mining Software Defects

As manual classification and detection of bugs does not scale to a large set of projects, we turn to data mining approaches such as the rule based analysis and the machine learning approach. In rule-based analysis [20, 23], programmers specify error patterns and coding rules manually, and then a software tool scans the source to check against them. The accuracy is comparably high and scanning speed is fast. However, as more libraries have been used, it has become difficult to specify rules because of the complexity and the quantity of the libraries. The machine learning approach [2] (MLA) uses data mining techniques to learn from the existing source code, and extract the knowledge in the form of coding patterns. This approach can characterize new patterns from newly created libraries automatically. However, MLA requires a large amount of training data.

Our approach mines usage patterns for specific library APIs (application programming interfaces) and locates the potential defects in software projects. An API in a library is designed to provide the client programs multiple working scenarios but an API used in unexpected scenarios may have unexpected results. Mining a huge amount of source code should enable us to extract the working scenarios, which we call the API usage patterns. As most APIs in released software are used correctly, the knowledge extracted is useful for detecting misused scenarios. This process is like extracting the knowledge from many experienced programmers to help the entire programming community. When pieces of source code violating the API usage patterns are found, the code should be checked for defects. We hope the process of detecting potential defects as described here can complement the current testing process practiced in the software assurance community.

3.1 System Overview

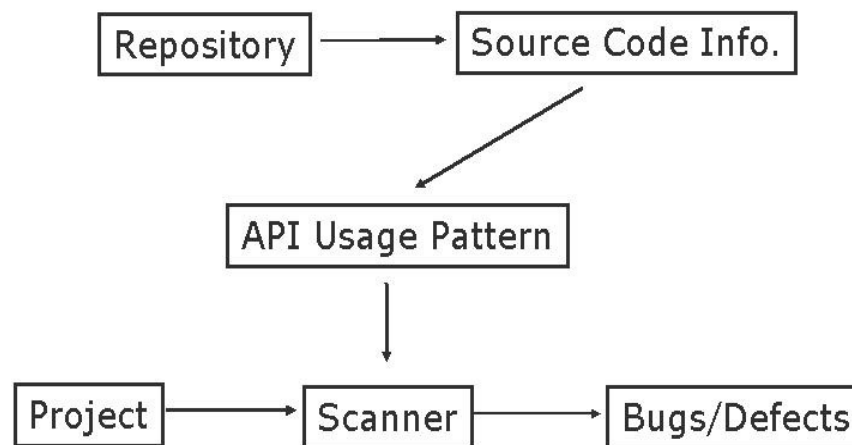


Figure 10 System Overview

Our approach in finding defects through API patterns discovery is given in Figure 10. First source files are extracted from a software repository. Then API usage patterns are examined by examining the method calls within a method. For example `java.util.Iterator.next()` and `java.util.iterator.hasNext()` appear in the same function together frequently. That implies that

these two methods being called together are considered to be a usage pattern. High confidence usage patterns are selected and we scan the source code with these patterns. The scanner flags potential defects when the source code violates the patterns.

By collecting data from publicly available source code repositories, e.g., sourceforge.net, we can decompose the code into 'transactions'. A usage pattern may hide in each transaction. Different kinds of learning approaches, like association rule learning and statistical classification, can be used to mine the API usage patterns from these projects. After scanning each transaction by usage pattern, we can locate the violations in the source code and inform the programmers.

3.2 Mining API Usage Patterns

Software developers are encouraged to reuse existing libraries through their public APIs. However, misunderstanding between the library documents and the programmers may lead to software defects. Some research [2, 11, 12, 15] has been done to understand the API usage and lower the risk of defects made by misusing the libraries. To evaluate the feasibility of data mining software defects, we mined a large number of Java projects and characterized their usage patterns. We adopted the association rule learning (ARL) in our approach as it is a well-established method for discovering the relationships between variables in a large set of transactions.

ARL was introduced by Agrawal [17] to mine the item relations in a database. The technique is composed of two steps. The first step is to find the item sets with a number of occurrences exceeding a threshold (called minimum support). The second step is to generate these association rules from these frequent item sets where the confidence exceeds the threshold. For example, if there are six transactions: {1, 2, 3, 4}, {2, 3, 4}, {2, 3, 4}, {1, 2, 4}, {2, 4} and {2, 3}, and the minimum support is 3, then we can find the following frequent sets: {2}, {3}, {4}, {2, 3}, {2, 4}, {3, 4} and {2, 3, 4}.

After frequent item sets are mined, the association rules can be generated. An association rule can be denoted as $A \Rightarrow B$ with confidence c and support d , where A and B are mutually exclusive subsets from a frequent item set. The support d is the occurrence of $A \cup B$. The higher the support value, the more common the usage pattern is found in the source. The confidence c is the probability that item set B occurs following the item set A . Note that it is possible that a pattern with high support and low confidence, for example A is a widely used API while B isn't, the support of $A \Rightarrow B$ may be high, but the probability that B following A is low.

In the association rule generation step, the confidence value is obtained by the ratio of $A \cup B$'s support and A 's support ($\text{conf}_{A \Rightarrow B} = \text{sup}(A \cup B) / \text{sup}(A)$). In the origin ARL algorithm such as Apriori [6], the size of suffix set in the rule is arbitrary. But we only generate those association rules $A \Rightarrow B$ for which subset B contains only one element. We can do that because the rule is used to detect violation in the form: " A occurs while B doesn't". This improvement reduces the time complexity from 2^n to n , where n is the size of a frequent set.

For example, suppose $B = \{b_1, b_2, \dots, b_n\}$. Source code violates the rule $A \Rightarrow B$ if and only if source code violates at least one of the following rules: $A \Rightarrow \{b_1\}$, $A \Rightarrow \{b_2\}$... $A \Rightarrow \{b_n\}$. Thus detecting violation of the single-suffix rule is completeness.

Usually only association rules with high confidence are kept. However, considering that an API may have several usage patterns, the association rules with low confidence may be useful to our bug detecting. If a rule with the same prefix whose confidence sum is greater than a desired threshold is found, this rule is kept as well.

For example, both $A \Rightarrow \{b_1\}$ and $A \Rightarrow \{b_2\}$ have a confidence 0.45. The confidence sum of these two rules is 0.9, which is a high confidence. Then these two rules are kept in the form $A \Rightarrow b_1 | b_2$. This rule can be interpreted as 'when A occurs, at least one of b_1 and b_2 occurs', which means the two use a scenario of the API.

```

public void write(byte[] outText) throws IOException{
    File f=new File("foo.txt");
    System.out.println("foo.txt");
    if(!f.exists()){
        FileOutputStream out=new FileOutputStream(f);
        out.write(outText);
        out.close();
    }else{
        System.out.println("File exists");
    }
}

```

Figure 11 A write method example in an ARL transaction.

We consider each member function of a transaction considered in the ARL approach. We ignore the rest of the source code information such as control flow, invoke dependency and parameters of the API. For example, the `write()` method in Figure 11 is converted to the transaction in Table 1.

Table 1 An ARL transaction derived from the write method in Figure 11

ID	API
0	java.io.File.File()
1	java.io.File.exists()
2	java.io.FileOutputStream.FileOutputStream()
3	java.io.FileOutputStream.write()
4	java.io.FileOutputStream.close()
5	System.out.println()

3.3 Experiment Results

We performed the experiments on the Apache software repository [22], a large open source application repository. A crawler was introduced to download 51 Java projects for our experiments. We arbitrarily divided them into two data sets, a small one containing 15 projects, e.g. Abdera, ActiveMQ and others and a large one containing 36 projects, e.g., Derby, Excalibur and others. We employed two data sets, as that would allow us to gain some insight into the impact from the data size. The number of lines of code (LOC), the number of source files, the number of Java classes, the total source file size in bytes and number of methods for each data set are given in Table 2. For the purpose within the scope of this paper, i.e., to discover patterns across projects, we focused on the library calls to `java.*` and `javax.*`. We discarded those methods that invoke `java.*` or `javax.*` just once.

Table 2 Summary of data set

Data Set	LOC	Source files	Java Classes	File size (bytes)	Number of methods
Small	4,119,645	27,618	28,154	144M	17,937
Large	15,714,997	103,545	105,117	601M	82,879

We chose the Apriori algorithm [6], a data mining algorithm for retrieving frequent sets from a large amount of transactions, in our experiment. We first conducted an experiment to understand the impact of maximum item set size on rule discovery by looking at the actual item set sizes discovered while keeping the max size set to 20. The results of the experiment are given in Figure 12. By keeping the max size to 20, we essentially captured all the rules.

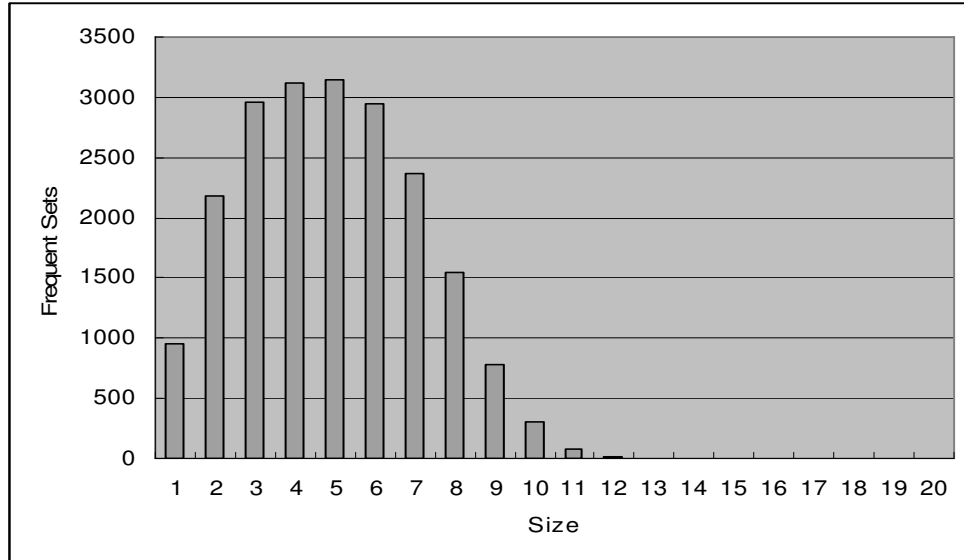


Figure 12 Frequent set distribution for the small data set. The minimum support count is 10.

We proceeded to the rule mining and verifying steps of the experiments using a Pentium D 3.0 GHz processor machine with 2GB memory. The rule-mining step took 35 minutes to extract 85,895 rules from small data set while it took 28 hours to extract 409619 rules from the large data set. The number of rules discovered by the selected minimum support, maximum rule size and confidence threshold are given in Table 3. The minimum support, the maximum rule size and the confidence threshold were chosen so a fair number of rules were discovered in each data set.

Table 3 Summary of association rule

Data Set	Minimum support	Maximum rule size	Confidence threshold	Number of rules
Small	8	20	0.85	85,898
Large	30	20	0.90	409,619

In the verifying step, we applied two sets of rules in ANT, which is one of the projects from Apache SVN. The results show that our method found 2664 and 2178 matched rules and 71 and 42 violated rules in ANT. It is difficult to classify if the violation is a defect or not. In order to measure the effectiveness of finding defects, we manually introduced several bugs by removing some method calls of `java.*` or `javax.*`, and applied rules again. The precision and recall of our method are listed in Table 4. The results show that using association rules learning under configuration above, about 29% and 27% of removed method calls are found, and 74% and 81% of new violations are caused by removed method call. While the results are promising, we agree that introducing bugs may not be the best approach. We hope to detect code defects by extracting code versions in the future.

Table 4 Rule verifying on ANT

Data Set	Matched rule	Violated rule	Inserted bugs	Recall (bugs)	Precision (bugs)
Small	2,664	71	696	0.29	0.74
Large	2,187	42	696	0.27	0.81

4 Discussion

Currently, item set mining [2, 6, 10] and sequence mining [7, 11, 12, 13] are the two primary kinds of data mining methods used to find software defect patterns. The items used as data sources for data mining methods are primarily function calls. Item set mining produces unordered patterns, i.e., sets of function calls, whereas, sequential pattern mining produces partially ordered patterns [14]. In the current literature,

most of the work in these data mining methods focuses on the missed function defect pattern described in section 2.5.2.

Two researchers [8, 9] tackle the issue of the neglected conditions. Both of them used some techniques of program analysis with data mining methods. Without program analysis, data mining methods suffer from issues of high false positives as their pattern elements are not necessarily associated with program dependencies [8]. There may be a trend combining data mining methods with program analysis techniques.

Table 5 shows the kind of defect patterns covered by current data mining methods. The first column is the defect classification used in this paper. The second column shows whether the corresponding pattern is covered by current data mining literature. The third column offers some typical work in the literature. The fourth column indicates whether program analysis techniques are used in the current literature to detect the corresponding defect pattern that is in the same row. And the last column is the defect ratio of each pattern in our Apache Harmony case study described in section 2.7.

From Table 5, we can see that current data mining methods can find missed function call patterns, which are mainly call pairs, call sequence and call structures. With the aid of program analysis, Neglected conditions can be addressed. Other kinds of patterns are not covered to the best of our knowledge in the literature.

In the Harmony's case, the missed function calls pattern related defects, which can be detected by stand-alone data mining methods, only constitute 6 percent. There are still large numbers of defects to be detected by data mining methods.

Table 5 Patterns covered by data mining methods

Pattern		Covered by literature?	Example Work	Using Program Analysis?	Defect % in Harmony case study
RM		No	-	-	25
VO		No	-	-	0
PC		No	-	-	12
CC	NC	Yes	[8, 9]	Yes	37
	Other	No	-	-	
DF		No	-	-	2
MF	CP	Yes	[2, 10]	No	6
	CSq	Yes	[11]	No	
	CSt	Yes	[13]	No	
	Other	No	-	-	
EF		No	-	-	2
Data use	VB	No	-	-	1
	VC	No	-	-	1
	VE	No	-	-	8

5 Summary

We analyzed the characteristics of code defects from the Apache Harmony project and found six general defect patterns: requirements engineering, value overflow, project configuration, conditions checking, function call usage and data usage. Three quarters of the defects come from requirements engineering, project configurations and condition checking, which are not detected by the data mining approach we developed. However, we still have a strong interest in using data mining techniques to identify software defects. We have shown that data mining methods can detect the missed function pattern, which constitutes 6% of all the defects in Harmony's case study. We believe as we learn more, and with the aid of program analysis, data mining methods should be able to detect other patterns.

Acknowledgments

The authors thank Bob Cohn, Ganesh Prabhala, Bob Scott and Khun Ban for their reviews that greatly improve the quality of this paper.

References

1. Kingsum Chow and David Notkin. 1996. Semi-automatic update of applications in response to library changes. In Proceedings of the 1996 international Conference on Software Maintenance (November 04 - 08, 1996). ICSM. IEEE Computer Society, Washington, DC, 359.
2. Livshits, B. and Zimmermann, T. DynaMine: Finding Common Error Patterns by Mining Software Revision Histories in Proceedings of 13th International Symposium on Foundations of Software Engineering (FSE'05) (Lisbon, Portugal, September, 2005), 296-305
3. M. Martin, V. B. Livshits, and M. S. Lam. Finding application errors and security flaws using PQL: a program query language. In Object Oriented Programming, Systems, Languages and Applications (OOPSLA'05), pages 365–383. ACM, 2005.
4. D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In Proceedings of the Fourth Symposium on Operating Systems Design and Implementation, pages1–16, 2000.
5. Ted Kremenek, Paul Twohey, Godmar Back, Andrew Ng, Dawson Engler. From Uncertainty to Belief: Inferring the Specification Within. Proceedings of the 7th Symposium on Operating System Design and Implementation, 2006
6. R. Agrawal, R. Srikant. Fast Algorithms for Mining Association Rules. Proc. of the 20th Int'l Conference on Very Large Databases, Santiago, Chile, Sept. 1994
7. R. Agrawal and R. Srikant. Mining sequential patterns. In Proc. 1995 Int. Conf. Data Engineering, pages 3-14, Taipei, Taiwan, Mar. 1995
8. Suresh Thummalapenta, Tao Xie. NEGWeb: Detecting Neglected Conditions via Mining Programming Rules from Open Source Code. North Carolina State University Department of Computer Science Technical report TR-2007-24, September 16, 2007
9. Ray-Yang Chang, Andy Podgurski, Jiong Yang. Finding What's Not There: A New Approach to Revealing Neglected Conditions in Software. In Proc. ISSTA, pages 163–173, 2007.
10. Z. Li and Y. Zhou. PR-Miner: Automatically Extracting Implicit Programming Rules and Detecting Violations in Large Software Codes. In Proc. FSE, pages 306-315, 2005
11. Tao Xie and Jian Pei. MAPO: Mining API Usages from Open Source Repositories. In Proceedings of the 3rd International Workshop on Mining Software Repositories (MSR 2006), Shanghai, China, pp. 54-57, May 2006
12. Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API Patterns as Partial Orders from Source Code: From Usage Scenarios to Specifications. In Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2007), Dubrovnik, Croatia, pp. 25-34, September, 2007
13. Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic. An Approach to Mining Call-Usage Patterns with Syntactic Context. ASE'07, November 5-9, 2007, Atlanta, Georgia, USA
14. Huzefa Kagdi, Michael L. Collard, and Jonathan I. Maletic; Comparing Approaches to Mining Source Code for Call-Usage Patterns. Fourth International Workshop on Mining Software Repositories (MSR'07)
15. Huzefa Kagdi, Shehnaaz Yusuf, Jonathan I. Maletic, Mining Sequences of Changed-files from Version Histories, MSR'06, May 22-23, 2006, Shanghai, China
16. Harmony <http://harmony.apache.org/>
17. R. Agrawal; T. Imielinski; A. Swami: Mining Association Rules Between Sets of Items in Large Databases, SIGMOD Conference 1993: 207-216
18. Geoffrey Phipps, Comparing observed bugs and productivity rates for Java and C. Software Practice and Experience, 1999, vol. 29. 345-358.
19. Purify <http://www.ibm.com/software/awdtools/purify/>
20. FindBugs. <http://findbugs.sourceforge.net>
21. Yoav Freund. An adaptive version of the boost by majority algorithm. In Proceedings of the 12th Annual Conference on Computational Learning Theory, 1999.
22. Apache SVN <http://svn.apache.org/repos/asf/>
23. PMD/Java. <http://pmd.sourceforge.net>
24. Jianyong Wang and Jiawei Han. BIDE: Efficient mining of frequent closed sequences. In Proc. 20th International Conference on Data Engineering, pages 79-90, 2004.
25. R. Chillarege et al: Orthogonal Defect Classification - A Concept for In-Process Measurements. IEEE Transactions on SW Engineering, vol. 18(11), 11/1992
26. Marek Leszak, Dewayne E. Perry, Dieter Stoll, A Case Study in Root Cause Defect Analysis. icse, pp.428, 22nd International Conference on Software Engineering (ICSE '00), 2000

Data Mining for Process Improvement

Paul Below

paul_below@qsm.com

Quantitative Software Management, Inc. (QSM)

<http://www.qsm.com/>

Biographical sketch:

Paul Below has over 25 years experience in the subjects of measurement technology, statistical analysis, estimating and forecasting, Lean Six Sigma, and data mining. He has provided innovative engineering solutions as well as instruction and mentoring internationally in support of multiple industries. He serves as services consultant for Quantitative Software Management (QSM) where he provides clients with statistical analysis of operational performance, helping strengthen competitive position through process improvement and predictability.

Paul is a Certified Software Quality Analyst, and a past Certified Function Point Specialist. He is Six Sigma Black Belt. He has been a course developer and instructor for Estimating, Lean Six Sigma, Metrics Analysis, Function Point Analysis, as well as statistics in the Masters of Software Engineering program at Seattle University. He is a member of the IEEE Computer Society, the American Statistical Association, the American Society for Quality, the Seattle Area Software Quality Assurance Group and has served on the Management Reporting Committee of the International Function Points User Group. He has one US patent and two patents pending.

Abstract:

What do you do if you want to improve a process and you have 100 candidate predictor variables? How do you decide where to direct your causal analysis effort? Similarly, what if you want to create an estimating model, and you have so many factors you do not know where to start?

Data mining techniques can be used to filter many variables down to a vital few to attack first, or to build estimating models to predict important outcomes.

In this paper, I provide specific software engineering examples in four data mining categories: classification; regression; clustering; association.

When creating a predictive model to understand a process, the primary challenge is how to start. Regardless of the variable being estimated (e.g., effort, cost, duration, quality, staff, productivity, risk, size, rework) there are many factors that influence the actual value and many more that could be influential.

The existence of one or more large datasets of historical data could be viewed as both a blessing and a curse: the existence and accessibility of the data is necessary for prediction, but traditional analysis techniques do not provide us with optimum methods for identifying key independent (predictor) variables from a large pool of candidate variables. Unfortunately, the Lean Six Sigma body of knowledge does not include data mining as a subject area.

Data mining techniques can be used to help thin out the forest, so that we can examine the important trees.

Copyright Quantitative Software Management, Inc.

Published at the Pacific Northwest Software Quality Conference 2009

Introduction

What do you do if you want to create an estimate and you have 100 candidate variables to use in your estimating model? Data mining techniques can be used to filter many variables to a vital few to build or improve model based estimates. Specific examples are provided in four categories: classification; regression; clustering; association.

When creating an estimating model, the primary challenge is how to start. Regardless of the variable being estimated (e.g., effort, cost, duration, quality, staff, productivity, risk, size), there are many factors that influence the actual value and many more that could be influential.

The existence of one or more large datasets of historical data could be viewed as both a blessing and a curse: the existence and accessibility of the data is necessary for prediction, but traditional analysis techniques do not provide us with optimum methods for identifying key independent (predictor) variables from a large pool of variables.

Data mining techniques can be used to help thin out the forest, so that we can examine the important trees.

What is data mining?

There are many books on data mining, and each one has a slightly different definition. The definitions commonly refer to the exploration of very large databases through the use of specialized tools and a process. The purpose of the data mining is to extract useful knowledge from the data, and to put that knowledge to beneficial use.

Data mining can be viewed as an extension of statistical analysis techniques used for exploratory analysis, incorporating new techniques and increased computer power. A free introductory data mining booklet can be downloaded from <http://www.twocrows.com/booklet.htm>. Other sources are listed in the References section.

There are a number of myths that have grown up regarding the use of data mining techniques. Data mining is useful but not a magic box that spits out solutions to problems no one knew existed. Still required for success:

- business domain knowledge
- the collection and preparation of good data
- data analysis skills
- the right questions to ask

Considering the purpose of starting to create an estimating model, this leads to the following statement:

The hard thing is not figuring out which algorithm to use,
the hard thing is to figure out what to do with the results.

Researchers have created a number of new data mining algorithms and tools in recent years, and each has theoretical advantages and avid proponents. However, for the purpose of getting started with estimate model creation, tool selection is not critical. The practical advice is to try as many of different techniques as possible, as the difficult time consuming task is data preparation. Refer to a list of tools in the References section.

Model creation challenges

People love to interpret noise. Regardless of what the data shows, the audience will offer theories to explain the causes for what is observed. If a graph shows that performance has improved, someone will offer an explanation for why that happened. If you tell the audience that the graph was upside down, and performance has actually decreased, just as quickly someone will propose a reason for why *that* happened.

Figure 1 is an image of random noise. If you stare at it long enough, you will start to see some patterns. People are pretty good at pattern recognition, even if no pattern actually exists. That is one reason why statistical quality control, data mining, and hypothesis testing are useful - to help us see whether the patterns we think we see are real or whether they could be explained by randomness alone. Another reason is to help us find patterns that are real but are difficult to see.

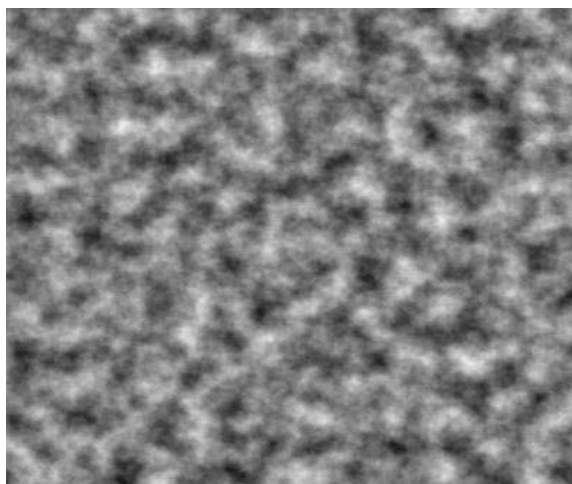


Figure 1: Random Noise

Exploratory analysis, including data mining, utilizes existing data that has already been collected. There are challenges with using such data, including:

- The databases already exist and almost always were created without considering analytical needs.
- Databases generally are built by committees, or have evolved from older systems through multiple stages. The variables stored include items that were used long ago as well as fields that someone thought might be useful someday, mixed in with data that are currently necessary. Many of the fields have values that are hard to decipher, or were used inconsistently by different populations of users.
- The structure of the data is often bad or the keys are not appropriate, making data extraction difficult.

Regardless of the data mining tools used, data extraction and validation is a major undertaking.

Once the data is extracted and placed in a readable format, the estimator is faced with dozens of input variables. Which of those variables should be used in the estimating model?

It is common for our variables to exhibit colinearity. Colinearity is when the variables are highly correlated with each other. In practical terms this means that those variables are measuring the same or similar things. Dumping all of these variables into a regression equation is not a way to receive a useful output.

Data mining can help us thin out the forest so that we can see the most important trees. Many of the data mining techniques can be used to identify independent variables that are influential in predicting the desired result variable. Success will depend more on the mining process than on the specific tools used.

Data Mining Models

“Statisticians, like artists, have the bad habit of falling in love with their models.” George Box

Data mining can aid in hypothesis testing as well as exploratory analysis.

There are many pure data mining products on the market, but they are typically very expensive. Some of the common techniques, however, are supported by basic statistical analysis tools which are much less costly. These techniques include all of the examples provided in this document. Examples of statistical analysis tools that support some or all of these functions are listed in the References section.

Data mining models can be placed into four categories as described in this table:

Category	Description	Purpose	Primary Data Type
Classification	Split the data to form homogenous subsets	Predict response variable	Discrete is best
Regression	Best fit to estimating model	Predict response variable	Continuous (ratio or interval)
Clustering	Group cases that are similar based on selected variables	Identify homogeneous groups of cases	Any
Association	Group variables that are similar	Determine colinearity, identify factors that explain correlations	Ratio or interval (not categorical)

Another facet of data mining models is whether they are white box or black box. Although black box techniques are often used for prediction (examples include neural networks and k-nearest neighbors), users generally dislike them because it is difficult to see how the model works.

White box techniques are often used for interpretation and, for the purpose of identifying key influential factors to create an estimating model, they should be used first.

Classification example

One classification technique is a tree. In a tree, the data mining tool begins with a pool of all cases and then gradually divides and subdivides them based on selected variables.

The tool can continue branching and branching until each subgroup contains very few (maybe as few as one) cases. This is called overfitting, and the solution to this problem is to stop the tool before it goes that far.

For our purposes, the tree is used to identify the key variables. In other words, which variables does the algorithm select first? Which does it pick second or third? These are good candidate variables to be used in an estimating model, since the tree selected them as the major factors.

In Figure 2, we see an example that started with a data set of 841 cases, taken from a database of client information. The cases were assigned to one of four groups based on user satisfaction, and in the top box each group is listed with the fraction of the cases. So, for example, group I contains 6.8 percent of the 841 cases.

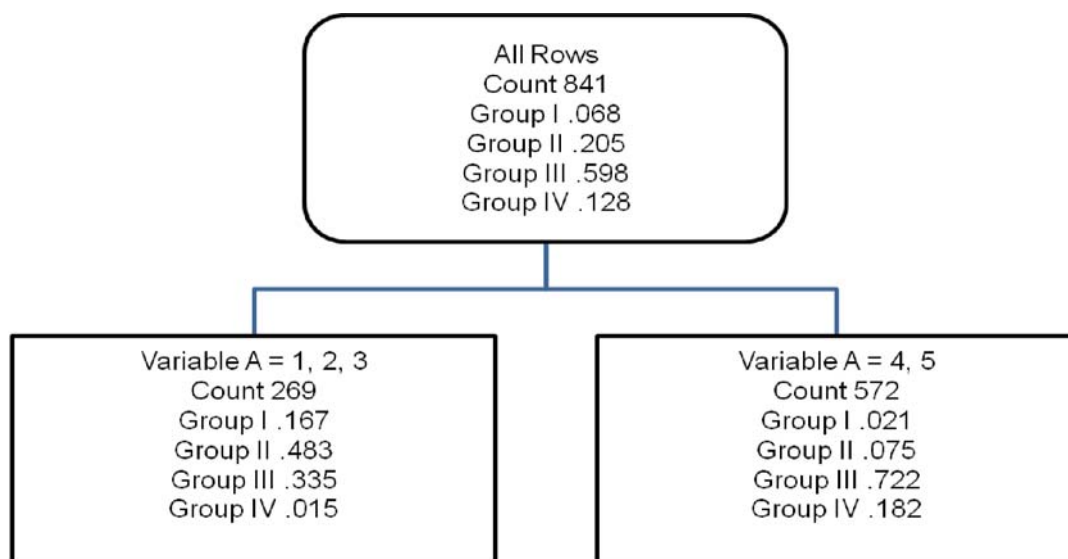


Figure 2: Tree Example

The tree algorithm examined all of the variables, and selected Variable A to be the first branch. Variable A has possible integer values from 1 to 5. As we can see, the algorithm put the cases where variable A is equal to 1, 2, or 3 in the left branch and those with Variable A equal to 4 or 5 in the right branch.

The left branch has 269 cases, including most of the cases in groups I and II, whereas the right branch ended up with 572 cases, including most of the cases in groups III and IV.

Variable A by itself is not a sufficient predictor of which Group a case will belong to, but the tree is telling us it is an important factor.

The tree would have additional branches, but Figure 2 is sufficient to aid in explaining how the tree is used.

Regression and Correlation examples

The data used in the remaining examples came from an industry data set. It is based on a sample of 193 projects extracted from the QSM database.

The output in the examples is for illustrative purposes and should not be used to reach conclusions about performance of specific software projects.

Stepwise regression is a type of multivariate regression in which variables are entered into the model one by one, and meanwhile variables are tested for removal. It can be a good model to use when supposedly independent variables are correlated. Stepwise regression is one of the techniques that can help thin out the forest and find important predictive factors.

Figure 3 is a summary output of a stepwise regression that went through nine steps to build the best model. The dependent variable being predicted was errors detected prior to deployment. The stepwise regression selected 9 variables that fit the threshold for inclusion, while excluding 20 other variables (not listed).

Model Summary

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
9	.840	.706	.691	330.332

Model		Sum of Squares	df	Mean Square	F	Sig.
9	Regression	47883321.563	9	5320369.063	48.757	.000
	Residual	19968796.365	183	109119.106		
	Total	67852117.927	192			

Predictors: (Constant), Effective SLOC, Life Duration (Months), MB Time Overrun %, MB Effort (MM), Life Peak Staff (People), Data Complexity, MBI, MB Effort %, Mgmt Eff.
Dependent Variable: Errors (SysInt-Del)

Figure 3: Regression Summary

The nine variables selected, in order, were: effective source lines of code; project life cycle duration in months; percent of duration overrun of Main Build (design through deploy); Main Build man months of effort; peak staff; data complexity; Putnam's Manpower Buildup Index; percent of effort expended in Main Build; Management effectiveness. Note that two of these nine variables (data complexity and management effectiveness) are qualitative, scored on a scale of 1 to 10 where 5 is average and 10 is high.

The first number to look at in figure 3 is the Sig in the rightmost column. The most commonly used significance threshold is .05, which means that the variable or model would be significant at the 95% level. In the example, the value .000 means that we have less than a 1 in a thousand chance of being fooled by random variation into thinking this model is significant.

Although all 9 variables selected are clearly significant, the overall model created has an adjusted R square of .691, which means that these 9 variables taken together are explaining about 69% of the variation in errors found. This

may not be the best model to use for estimating, but it is important to look at each of the nine variables if the intent is to create an estimating model or if we need to reduce the number of errors found in the future.

The coefficients of the stepwise regression formula are displayed in Figure 4. Each variable is listed next to the coefficient B, which is the multiplier in the linear equation.

Coefficients: Dependent Variable: Errors (SysInt-Del)

Variable	Unstandardized Coefficients		Sig.	95% Confidence Interval for B	
	B	Std. Error		Lower Bound	Upper Bound
(Constant)	-580.411	239.656	.016	-1053.255	-107.568
Effective SLOC	.001	.000	.000	.001	.001
Life Duration (Months)	27.633	5.832	.000	16.126	39.139
MB Time Overrun %	.026	.006	.000	.015	.037
MB Effort (MM)	1.535	.326	.000	.892	2.177
Life Peak Staff (People)	-7.438	1.905	.000	-11.197	-3.679
Data Complexity	66.840	18.269	.000	30.795	102.886
MBI	33.683	14.609	.022	4.859	62.507
MB Effort %	3.924	1.552	.012	.862	6.987
Mgmt Eff.	-50.012	22.775	.029	-94.948	-5.076

Figure 4: Regression Coefficients

The equation that yielded the adjusted R square of .691 is:

Errors = -580 + (.001*ESLOC)+(27.6*Duration)+(.026*overrun%)+(1.5*MB Effort)-(7.4*peak staff)+(66+data complexity)+(33.68*MBI)+(3.9*MB effort %)-(50*Mgmt Eff)

The factors in the equation can be determined from reading the numbers in the B column.

A negative number means a negative correlation. One counterintuitive result of this example is the coefficient for peak staff. The negative coefficient means in this model the larger the peak staff the smaller the number of errors detected. This type of result is why it is necessary to evaluate the data in more depth and do additional analysis before using the model. Sometimes, negative correlations are expected. For example, Management Effectiveness has a negative coefficient meaning that a higher effectiveness results in a lower number of errors.

The two rightmost columns, the 95% confidence intervals, are useful as an indication of the uncertainty in the coefficients. The lower and upper bound for any variable should not straddle zero. If it did, that would be an indication that we lack confidence in the factor B. Another method is to compare the value of the standard error to the value of the coefficient; ideally the standard error should be much smaller than the coefficient B. Also, the Sig should be small, ideally less than .05.

In addition to regression, correlation can be used to identify candidate important variables. This can be done by selecting the dependent variable first for the correlation and then the list of independent variables. There are different types of correlation that can be used. For ratio data Pearson correlation can be used. For ordinal data, Kendall's Tau-B will work. For nominal (categorical) data, a chi square test can be used on a crosstab (two way table) to determine significance.

It is important to note that these tests will determine linear correlations. Sometimes correlations exist but are nonlinear. One technique for exploring those relationships is transformation, which is not discussed in this paper.

Clustering example

Cluster techniques detect groupings in the data. We can use this technique as a start on summarization and segmentation of the data for further analysis.

Two common methods for clustering are K-Means and hierarchical. K-Means iteratively moves from an initial set of cluster centers to a final set of centers. Each observation is assigned to the cluster with the nearest mean. Hierarchical clustering finds the pair of objects that most resemble each other, then iteratively adds objects until they

are all in one cluster. The results from each stage is typically saved and displayed numerically or graphically as a hierarchy of clusters with subclusters.

Figure 5 is the output of a K-Means example run from the QSM sample with the output set to create exactly three clusters.

The tool placed the largest projects in the first two clusters. These projects had more errors, more staff, and higher productivity than the third cluster. One difference between the first two clusters is that the projects in the second cluster tended to have poor estimates of effort.

Final Cluster Centers

	Cluster		
	1	2	3
Project Count	5	22	166
Life Effort (MM)	750.7	617.8	89.1
Errors (SysInt-Del)	1898	1030	186
Errors First Month	138	117	8
Total FP	37167	26533	2648
Effective SLOC	1272194	298791	26444
Life Duration (Months)	21.3	18.4	9.3
Life Peak Staff (People)	56.5	61.1	15.4
Life Avg Staff (People)	23.8	26.5	7.1
MB Eff Overrun %	.0	62.0	45.8
SLOC/MB MM	2384.5	1606.4	910.9
Putnam's PI	24.4	21.5	14.1

Figure 5: Cluster Example

We may want to stratify the projects into groups based on the above distinctions prior to conducting additional analysis. This may result in the need for more than one estimating model, or more than one process improvement project.

Association example

Association examines correlations between large numbers of quantitative variables by grouping the variables into factors. Each of the resulting factors can be interpreted by reviewing the meaning of the variables that were assigned to each factor. One benefit of Association is that many variables can be summarized by just a few factors.

In the following example using QSM sample data, Principal Components analysis was used to extract four components. The Scree Plot in figure 6 was used to determine the number of components to use. The higher the Eigenvalue, the more important the component is in explaining the associations. Selection of the number of components to use is somewhat arbitrary, but should be a point at which the Eigenvalues decline steeply (such as between components 2 and 3, or between 4 and 5). It turned out in this example that the first four components account for roughly half of the variation in the data set making four a reasonable choice.

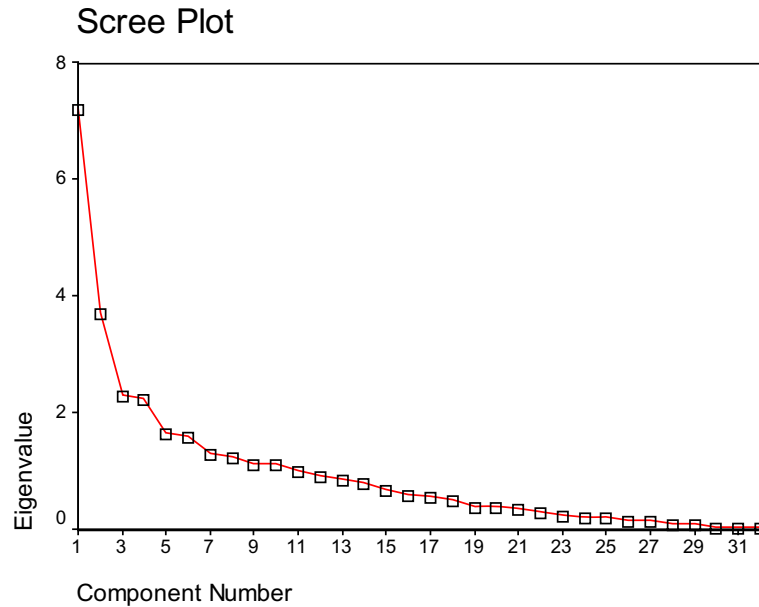


Figure 6: Scree Plot for Association example

Figure 7 displays variables with the most significant output for each component. The important numbers in the table are those with relatively large absolute values and have been shaded for easy reference.

- Component 1 is composed of a market basket of variables related to effort and size.
- Component 2 grouped variables related to the development team: knowledge; turnover; skill.
- Component 3 isolated the Manpower Buildup Index, which is the speed at which staff is added to a project.
- Component 4 linked the percent of effort expended in functional requirements to the percent expended in main build (design through deploy).

Variables that are seen to be closely related should be combined (or one should be chosen as the representative) as an input variable when creating prediction models or identifying root causes.

	Component			
	1	2	3	4
Life Effort (MM)	.920	-.152	.196	-.006
Effective SLOC	.652	.111	-.475	.106
Life Duration (Months)	.658	-.198	-.429	.066
Life Peak Staff (People)	.865	-.115	.338	-.137
Life Avg Staff (People)	.823	-.157	.381	-.156
FUNC Effort (MM)	.880	-.169	.151	.098
MB Effort (MM)	.925	-.160	.065	-.122
Func Effort %	-.241	.088	.236	.719
MB Effort %	-.059	-.072	-.247	-.765
Knowledge	.186	.770	.161	-.076
Staff Turnover	.083	-.717	.049	.110
Dev Team Skill	.133	.746	.029	-.225
MBI	-.006	-.011	.640	-.200

Figure7: Association example output

Summary

Once data has been collected and validated, the hardest work is behind you. Any data mining tools that are available to the researcher can be used relatively quickly on clean data. These data mining techniques should be used to filter an overwhelming set of many variables down to a vital few predictors of a key output (for example, quality).

Determination of the vital few is a key component of process improvement (such as Six Sigma projects) activities as well as prediction. With those key drivers or influencers of quality in hand, improvements can be designed and implemented with fewer iterations, effort or time.

In addition to process improvement activities, we use the “vital few” to build error prediction models, and then use the models to tune parametric project estimates for specific clients. The project estimate and plan is thereby not only an estimate of duration and cost to complete construction, but also includes the prediction of when the system will be ready for prime time.

References

Data Mining Websites:

- www.twocrows.com
- www.kdnuggets.com
- www.datamininglab.com

Data Mining Tools:

- Statistical tools that do some data mining techniques include: SPSS; SAS; JMP; SPlus; Minitab
- Specialized data mining tools include Salford Systems CART and MARS; SAS Enterprise Miner; PASW Modeler (SPSS); Insightful Miner (SPlus)

Books:

- *Introduction to Data Mining*, by Pang-Ning Tan, et al, Addison-Wesley, 2006.
- *Principles of Data Mining*, by David Hand, Keikki Mannila and Padhraic Smyth, MIT Press, 2001.
- *Data Mining – Concepts, Models, Methods and Algorithms*, by Mehmed Kantardzic, John Wiley and Sons, 2003.
- *Data Mining: Opportunities and Challenges*, by John Wang, IDEA Group, 2003.

Sources of industry metrics:

- For practicing the techniques described, a self selected database of project metrics can be purchased from <http://www.isbsg.org/>
- QSM maintains a project database and coordinates a benchmarking consortium: <http://www.qsm.com/>

Distributed Team Collaboration

Kathy L. Milhauser
kathym@pdx.edu

Abstract

Working in distributed teams is becoming increasingly common as companies extend and diversify their operations across geographic boundaries. Learning to form and sustain high-performing distributed teams with members in multiple locations, time zones, and representing diverse cultural perspectives requires new skills and new approaches to project team collaboration. This paper will outline the drivers causing this transformation in project work, introduce a variety of models that represent distributed team configurations, summarize some of the challenges inherent in leading and contributing to distributed teams through a set of case studies, and suggest practices that are emerging to optimize distributed team performance.

Biography

Kathy Milhauser has worked in Information Technology and Human Resources Management for over 20 years, with specific experience in product development, manufacturing, supply chain, and marketing for a Fortune 500 company, as well as in government and health care industries. Kathy is currently a Program Director at City University of Seattle where she manages programs in Project and Technology Management online and internationally. Kathy is also an adjunct faculty member of the OMSE program, where she teaches the Professional Communication Skills and Distributed Software Team Collaboration courses. Kathy is a doctoral student at George Fox University, where her research is focused on globally distributed team challenges and best practices. Her work has appeared in the Software Association of Oregon newsletter and in a book on Managing Learning in Virtual Settings.

Introduction

In the global business environment, every organization, team, and individual has the potential to compete for opportunities. What Friedman refers to as the flattening of the world has truly created a level playing field for many industrious individuals, teams and organizations (Friedman 2007). It is not surprising that the trend toward distributed organizations has increased in recent years. Organizations of all sizes and in a wide range of industries have found ways to cross geographic boundaries. This broadening of scope and focus is allowing large and small organizations to expand from a local and regional perspective to a global one. Change drivers such as developing economies, increased competition, and changes in political and trade relationships have facilitated an increase in the ability of organizations to globalize their operations just as advances in technology have accelerated the pace of global connectedness.

Globalization not only affects how organizations interact with each other, but even how they interact internally, as distribution of the organization creates new workgroups who must learn to work together effectively. As organizations diversify and distribute, so do the belief systems, values, and eventually the behaviors of their members. While maintaining a focus on an organization's mission and goals is challenging for all organizations, it is especially daunting when the workforce becomes dispersed in multiple locations. This can create tension between local perspectives and those of the overall organization as remote teams form their own sense of identity.

The distributed team phenomena is both born from opportunity and created out of necessity. On the one hand, organizations are distributing their operations because they *can*. Permeable economic boundaries, trade agreements, and maturing technologies have made distribution profitable. On the other hand, organizations are distributing their operations because they *must* in order to compete in this new landscape where every organization, team and individual has just become empowered to compete. As this trend will no doubt continue to accelerate, now is a good time for organizational leaders to develop plans for how to lead distributed teams effectively.

1.1 Background

Research into team development models has its roots in the study of sociology and the formation of group norms. The idea that there might be a science to organizing the work of individuals into teams reaches as far back as the early 1900s when Frederick Taylor published his research on scientific management methods (Taylor 1911). What is unique in recent years is the ability of individuals to work in teams without being in the same physical location or even time zone. This presents a variety of obvious challenges, such as communication, coordination of tasks, and adapting to cultural differences. New opportunities for collaboration are just now coming into focus as researchers look at this emerging phenomenon.

Martinelli, Raschulte, and Wadell refer to a trend toward individuals becoming global managers by accident as their organizations distribute operations (Martinelli, Raschulte, and Wadell, 2009). Managers in this situation typically find themselves responsible for individuals that are broadly scattered around the globe in an often loosely coordinated fashion. As these managers interact with their teams, they are faced with the need to develop new approaches to leading their teams effectively in the distributed environment.

Deborah Ancona suggests that traditional team development practices are insufficient when applied to the distributed team (Ancona 1990). Ancona reviewed a study of 100 sales teams and the notable difference between perception of team satisfaction and actual team performance. The teams in the study that were focused on internal efficiency and experienced the most harmony in their teamwork were not the ones performing at the highest level. The highest performing teams were those with an external focus on their customers, suppliers, and industry, often resulting in high degrees of creative tension inside the team, but also opportunities for more innovative ideas. The external focus of the highest performing teams raises the question of whether the traditional focus on internal efficiency is

not only inadequate, but also potentially limiting to the distributed team. This work resulted in the assertion that the evolving distributed team environment with a flattened structure and broader range of cultural and geographic diversity calls for a new team leadership model.

1.2 Distributed Team Models

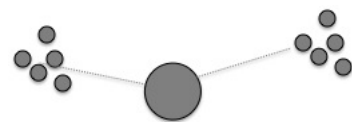
The research to support this paper was conduct in a variety of industries, including product development, marketing and sales, technology engineering and manufacturing, and higher education. Insights for this paper were gathered through semi-structured interviews and observations. All of the participants were members of global organizations with offices in multiple locations in the U.S., Asia, Europe, and the Americas. While the industries were varied, the challenges were found to be quite common.

The term “distributed teams” is used loosely to describe any situation in which people are working without being in the same physical location. While this is a good general description, it can also lead to confusion related to several key elements that distinguish distributed team behavior. For the purposes of this paper, distinctions will be made in three areas when describing distributed team models: the basic organizational structure (central or distributed); the primary team focus (internal or external); and the primary information flow experienced by team members (organization-to-team, organization-to-individual, or team-to-team). These distinctions are depicted in the following models. While still very general, these models aid in differentiating between three unique configurations commonly found in distributed team interaction.

	Decision Model	Team Focus	Information Flow
#1 – <i>Central / Remote</i>	Central	Internal	Organization-to-team
#2 – <i>Distributed Teams</i>	Distributed	External	Team-to-Team
#3 – <i>Distributed Employees</i>	Central	Internal	Organization-to-individual

Regardless of the exact configuration of the distributed team, some challenges are common whenever people are working across distance. Clearly it is more difficult to communicate and coordinate tasks when individual employees or full teams are not located in the same place and able to work face-to-face. Further, people working outside of the organization on distributed teams or as remote employees tend to miss a lot of informal, contextual information that aids other employees in making sense of the information they do receive.

1.2.1 Model #1 – Central / Remote



The first model is not a new one, but has been the default approach organizations have taken to developing presence in local communities. It is especially common for global organizations to create remote teams in local markets or to take advantage of local labor. What is changing is the trend toward creation of remote teams for purposes of employing highly skilled talent from geographies remote to the central organization, as is seen in the outsourcing and off-shoring trends in recent years. In this model, organizational control is still centralized. Strategy is set by a central organization with varying degrees of engagement with local offices and remote teams. For

the most part, the remote team is focused on executing a central strategy while providing input from their local perspective. As this model is often used in large global companies as they distribute operations to local markets, the issue of coordination and alignment is often stated as one of the leading concerns.

As organizations become increasingly global and distribute their operations, teams, and individuals outside of the central location, there is danger of disconnects forming between the organization's mission, vision, values, and goals and those of the distributed team. The central organization runs the risk of becoming an "outsider" from the perspective of the remote team. This is a common phenomenon in remote offices that grow to resist and resent the "headquarters" mentality. Teams performing in remote markets have opportunities to bring their local insights back to the central organization and ensure that decisions consider local perspectives. Doing this, however, requires a sense of "oneness" that extends beyond the central organization into all of the locations where its operations have been distributed.

Another fundamental issue facing many central/remote teams is that of diversity and cultural awareness. This takes on two dimensions as teams experience higher levels of diversity in national culture and geographic location, and also as teams find themselves further and further from the "mother ship" organization that provides their foundational sense of meaning in their work. Appreciating different perspectives on the distributed team becomes a necessity to support understanding and cohesiveness on the team.

Challenges:

- Us vs. them mentality;
- Remote group isolation;
- Cultural differences;
- Risk of remote team disconnecting from organizational goals.

The Case Study in Global Product Development is an example of a Model #1 team configuration and provides some best practices that one organization developed to address their challenges and enhance team performance.

1.2.2 Model #2 – Distributed Teams



The second model is one in which the organization is becoming fully distributed, with complete teams performing outside of the central organization. The focus in this case is on coordination in a point-to-point model between teams. The model connects two or more teams from the same or different organizations in their efforts to collaborate in a distributed fashion. In this case, decision-making is often distributed to team leaders, with higher degrees of authority at the team level.

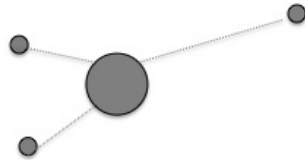
The challenge becomes one of internal integration between the distributed team and the central organization. As authority, autonomy, and empowerment are experienced in the distributed team environment, the central organization must find ways to connect the distributed team to the central mission of the organization in a cohesive and meaningful way. Motivation and extension of organizational culture become the prominent challenges experienced by organizations evolving their distribution model to this extent.

Challenges:

- Coordination of tasks without centralized support;
- Inefficiencies due to lack of clear leadership;
- Cultural differences;
- Conflict management;
- Risk of teams developing separate agendas and goals.

The Case Studies in Technology Teams provide examples of the Model #2 team configuration.

1.2.3 Model #3 – Distributed Employees



The third model is one in which a central organization has a number of distributed employees. This model is becoming increasingly common as organizations begin to allow employees to telecommute and strive to find uniquely skilled employees who do not reside in a central location. An example of this model would be an organization in a large metropolitan area that has decided to allow employees to telecommute. This can be a positive benefit for the employee who avoids the time and expense of commuting to a central work location. This model can also benefit the organization by reducing facility expenses and broadening the geographic area from which they can recruit talented workers. However, this model introduces new considerations for managers who are responsible for the work of the distributed employees.

Challenges:

- Supervision of work;
- Maintaining quality standards;
- Ensuring work is distributed equitably;
- Motivating distributed employees;
- Maintaining a sense of teamwork;
- Avoiding feelings of isolation on the part of the employee;
- Maintaining effective communication;
- Sustaining focus on the organization's goals.

Organizations experiencing success with the distributed employees model have found that a focus on human resource policies specific to telecommuting employees can ensure that work requirements are clear. Some organizations are beginning to develop telecommuting contracts with employees and specific training for managers in how to manage employees that are not in a central location. While this model will not be explored in depth in this paper, it is important to acknowledge that distribution of individual employees introduces similar management complexities.

2.1 Case Study – Global Product Development

Interviews with the members of a single product development team were conducted to support this case. The team was made up of central and remote components (Model #1 in this paper). The central portion of the team was located at the company's headquarters in the U.S. The remote portion of the team was located in Taiwan. The team was responsible for developing new products for markets around the world, with design and marketing headquartered in the U.S. and technical development

and manufacturing managed in Taiwan. The organization was attempting to shorten the time it takes to bring a new product to market, and had decided to use Lean process improvement methods to optimize the team's work. The team in this case was selected to pilot new methods for process improvement.

2.1.1 Face-to-Face Meetings:

The leaders of the global product team emphasized the power of interpersonal interaction in face-to-face settings. The organization launched the process improvement effort by bringing the full team together for a kick off meeting and the development of the project plan. By engaging all team members in a short workshop, this organization was able to not only build teamwork but also facilitate solid understanding of each role in the process. The workshop activities focused on where information handoffs would occur, what each person's responsibility in the project would be, and on developing agreements related to communication and quality standards. Integral to the design of this team workshop was time embedded for informal activities as well. This allowed the team members to get to know each other and build rapport before they were faced with the challenge of distributed teamwork. After the workshop, the team met periodically via video and audio conferencing, while continuing most of their work in a distributed fashion.

2.1.2 Managing Cultural Differences:

The global product team was made up primarily of American and Taiwanese members. Additionally, there were members from the Europe marketing team involved in the face-to-face workshop at the beginning of the project. This created an environment that combined individuals from some of the most direct and most indirect cultural extremes in Hofstede's cultural mindsets model (Hofstede 1980). Direct cultures are those where people tend to say exactly what they think and welcome conflict as a way to ensure that issues are addressed openly. Indirect cultures are the opposite – direct comments are seen as rude and avoided in order to ensure that no one involved loses face over any issues. Instead of open debate, most individuals from indirect cultures (typical of most of Asia) would choose to settle differences quietly so that no one would be embarrassed. Another significant difference in cultural mindset was in the dimensions of individualism and collectivism. The Dutch and Americans tend to be very individualistic – viewing achievement and responsibility from an individual perspective. The Taiwanese, like many Asian cultures, are much more collective in their orientation. They think first of the accomplishments and responsibilities of the group that they are part of, and tend to defer their own personal needs for the needs of the group or organization.

Differences in cultural mindset became apparent in the first workshop that this global product team conducted. The Dutch marketing executive was very adamant about what kind of product was needed for the Dutch consumer market, and was critical of the prototypes he had received from Taiwan. The American design team had been sending designs and design changes to Taiwan, and had not received any questions or feedback on the designs. Instead, the Taiwan team was doing their best to create the prototypes with the information they were provided. In the workshop, the team was instructed to diagram their process, with all of the handoffs between the different individuals on the team. Each team member was given a stack of post-its and a pen and they worked in small groups to document all of their activities. When the activities were all documented and visible on the process map, it became apparent that there were many activities happening in the Taiwan office that the Americans were not aware of. Curious about this extra work, the workshop facilitator asked one Taiwanese team member how often she received all of the information she needed to create a quality prototype: "How often is the information you receive 100% complete and accurate the first time you receive it?" The Taiwanese woman was silent. Eventually, with encouragement from her local Taiwanese team members, she was handed a pen and on the process map she wrote "0". This was the first time that the Americans on the team had been told that the designs and details they were sending to Taiwan were insufficient. It became a pivotal moment in the workshop and in the months of work to follow. The American team members learned to ask the Taiwanese what information they needed, and to purposefully confirm that all information sent to the remote Taiwan team was

complete and accurate. The Taiwanese team members, in their dedication to the collective needs of the full team, were willing to respond to direct questions and eventually an open dialogue ensued.

2.1.3 Sustaining Organizational Culture:

The organizational culture in this case provided support for the multi-cultural, distributed team effort. The organization has a history of grounding its employees in a central mission and value system. Creating strong and emotional messages that are clearly linked with the organization's cultural heritage supports this strategy. The organization has been in business for just over 35 years, and has grown from a small start-up in its industry to a globally recognized industry leader. A key differentiator for the organization from an organizational culture perspective is that its external brand message and internal value system are in close alignment. Examination of artifacts from the company's history reinforces the power of the internal and external messaging and provides evidence that employees have engaged deeply with the company's mission since the beginning of its existence. Further, conversations with highly tenured employees (some of them still working at the company since the beginning) reinforce that strong emotional ties to the heritage and purpose of the company have motivated them through many difficult times and periods of uncertainty.

In addition to the strong sense that the organization prizes individual achievement, there is evidence that team and collective achievement is valued. In interviews with participants, a tension could be detected between the values of individualism and collectivism. The idea of a high performing team, with individuals striving to achieve their personal best with the support of their team, emerged as the resolution to this tension. Even team members from the U.S. culture (which tends to be individually oriented) were able to accept that individual performance depended on the support of a team. In a sense, the participants viewed the team metaphor as a way to achieve their individual goals. Individuals on these teams reported the expectation that team members would challenge them personally through internal team competition, as well as support them toward the achievement of personal goals. Conversely, the Taiwanese members (with a stronger collectivist orientation) could accept the individualist orientation of the U.S. members when they understood how individual performance supported collective team goals.

When organizations with strong grounding in their cultural heritage, mission, vision, and values distribute their employees across distance, they create an environment where the organizational culture provides a foundation of support. This is especially important in remote offices where the national culture may be very different than that in the home office.

2.1.4 Summary/Lessons Learned:

- Face-to-face meetings are critical to building relationships, especially in multi-cultural teams;
- Clear understanding of roles, responsibilities, and integration of tasks can enhance the performance of a distributed team;
- Cultural differences are important to recognize, and can be leveraged if they are understood and accepted;
- Organizational culture can provide a common context for distributed teams if the mission and goals of the organization are well understood and interpreted in culturally relevant ways.

2.2 Case Studies – Technology Teams

The following mini-cases were collected from a variety of experiences working with software engineering students in the OMSE program. Technology teams have been experimenting with distributed team models for several years, since the Internet has made it possible for communication and collaboration to occur across distance. For this reason, technology teams tend to focus more on the tools available to support their tasks, and less on the practices that support effective teamwork. However, recent discussions with technology professionals indicate that issues of teamwork, communication, conflict management, and cultural sensitivity are coming into focus as technology teams develop their competency for distributed operation. The amount of experience the technology industry has with

distributed teams has also allowed for best practices in fully distributed teams to begin to emerge first in these teams. Although the companies in these cases have central / remote team models in place, they are also beginning to launch teams that are fully distributed with point-to-point interaction, as in Model #2 – Distributed Teams.

2.2.1 Stakeholder Management:

A distributed project team leader working in the technology manufacturing industry echoed the importance of focused and deliberate communication, emphasizing that communication inside the project team is not enough. Intentional, focused, and proactive communication with stakeholders becomes essential when working in a distributed environment. The project leader agreed that face-to-face meetings to build the team are essential. However, the team found it could not always rely on face-to-face meetings to communicate status and share documents with stakeholders. One specific incident occurred when the team set up an online meeting to share status and gain approval to move ahead on their project. The team had painstakingly updated their Google Docs site with the status report, links to technical specifications, process flowcharts, and slides for the meeting. They sent out the meeting invitation via Outlook, and instructed participants to be available on Skype at meeting time. When the meeting time arrived, all of the distributed team members were online and ready. The major stakeholder, however, was having trouble getting his computer connected to the network. When he resolved that issue, he couldn't locate the link to Google Docs. By the time the stakeholder was prepared to listen to the team's status, half of the meeting time had elapsed. The team was not able to meet their meeting objectives. The team learned that they needed to pay more attention to preparation for their meetings to ensure their stakeholders had access to necessary documents.

Individuals working in distributed teams indicate that planning and executing a successful online meeting takes much more time and focus on detail than a typical meeting where everyone is in the same room. Ensuring that everyone has access to the documents, that the online meeting tools are used effectively, and that there is adequate understanding of the issues and information being discussed becomes a key focus area for distributed team leaders. A "push" model with the documents being shared through a screen sharing tool became necessary for the team in this case. After the failed meeting, they learned to take steps to ensure that access to information was easy and seamless for all participants.

2.2.2 Practices Before Tools:

When many of us think of the challenges of distributed teams, we turn quickly to the available tools to see how to meet these challenges. Tools are sometimes selected before we fully understand the different practices involved in working in the distributed environment. Two graduate students in an online software engineering course were assigned to develop a proposal for online collaboration. Their goal was to convince their instructor to adopt new technologies for the program. The instructor was hoping to discover new tools to support software engineering practices. These students examined tools for software configuration management (SCM) and found quickly that it was necessary for them to step back from the tools and think first about what practices needed to change in order to interact with this process in a distributed fashion. Only then could they compare the potential tools for their assignment to emerging best practices in the SCM process. The students ended up spending more time introducing their instructors to newly emerging software configuration management processes than new tools.

2.2.3 New Tools and Practices Emerging:

The two graduate students discovered not only that there are new practices emerging to support distributed teamwork, but that tools are also quickly evolving to support the practices. In some cases, they learned about a potential change in practice when they discovered it by examining a tool. For example, because of the distributed nature of their SCM process, the potential for individuals to work offline as well as remote from the central code repository is creating practices for use when developers are "tethered" vs. "untethered." In other words, because distributed software engineers

are often working offline at a remote location, the tools are being evolved to exploit the fact that not all users are online at the same time. One practice the two graduate students discovered was that one tool would allow for “shelving” of code while working on other code. This practice emerged because it was possible in the tool, and was possible in the tool because the users are typically distributed as well as working in an offline, untethered fashion.

2.2.4 Online Simulating In-person:

Another example is the Wave tool recently introduced by Google for integrating email, instant messaging and wiki functionality in an interactive experience. Not only has Google figured out how to make these functions interact with each other, but they have also allowed users to pick and choose whom they share their commentary with. The group email then becomes analogous to a room of people, with whispered conversations on the side that not everyone can hear - all of this while remaining “in the room” and present to the larger conversation. Another advantage is that the dialogue in Wave is recorded so that it can be played back later, bridging the gap between synchronous and asynchronous experiences. Like the SCM example, in this case a tool is allowing practices to emerge in online interaction that are possible *only* because the participants are distributed.

Innovations like these are happening because they are technically possible, but more importantly because distributed employees and teams are in need of new ways to interact in more socially rich ways. The need for interpersonal connections drives the demand for tools that allow us to feel like we are together, even when we are working in distributed locations.

2.2.5 Summary/Lessons Learned:

- Occasional face-to-face meetings help to build and sustain relationships;
- Online meetings require more planning and preparation in order to be successful;
- Processes and workflows need to be clear before adopting new tools;
- New practices and tools are emerging quickly to support distributed work;
- Use of new tools introduces complexity as well as new practices to team processes.

Conclusion

The notion that individuals and groups of people do not necessarily have to be in the same place at the same time in order to perform work is no longer a new idea. Organizations in virtually all industries are recognizing this potential and beginning to experiment with how to distribute work to leverage opportunities and solve logistical problems. What is just now emerging is not the distribution of the workforce – that has been happening gradually over the past decade and much more rapidly in the last few years. What is new is the dawning realization that new ways of interacting, communicating, understanding each other's needs and perspectives have the potential to optimize organizational performance. Distributed teams can be either much less effective or much more effective than traditional teams. The key lies in looking first at the interpersonal needs and behaviors that emerge as people begin working in new team configurations, then looking at potential new practices and tools to support and empower new ways of working.

Moving to an Agile Testing Environment: What Went Right, What Went Wrong

Ray Arell

ray.arell@intel.com

Abstract

About two years ago, Ray Arell called his software staff together and declared, “Hey! We are going Agile!” Ray read an Agile project management book on a long flight to India, and, like all good reactionary development managers, he was sold! Now, two years later, their Agile/Scrum process has taken shape; however, its adoption was not without strain on development, test, and other QA practices. Join Ray as he takes you on a retrospective of what went right and, more importantly, what went wrong as they evolved to a new development/test process. He introduces the software validation strategies developed and adapted for Scrum, explains what makes up a flexible validation plan, and discusses their iterative test method. Learn how they use customer personas to help test teams understand expectations for quality in each sprint and employ exploratory testing in the Scrum development flow. If you see Agile in your development future, come discover what you’re in for, traps to avoid, and how to be successful. If you’re not ready for Agile, you’ll learn some new approaches that can be applied to traditional processes.

Biography

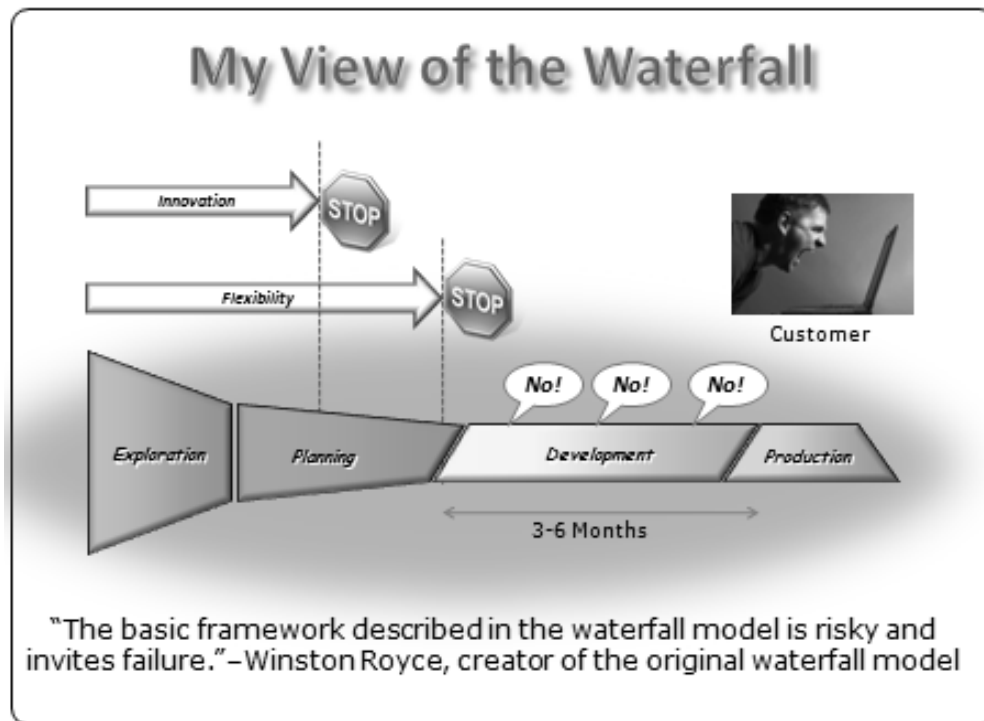
Ray Arell is a senior engineering manager and agilist at Intel, he has more than 20 years of hardware and software development, validation and management experience. During his tenure at Intel, he has worked on a variety of teams focused on CPU, chipsets and graphics system-level testing. Today, Ray manages an Agile software engineering team in Intel's Digital Office Platform Division and is a leading force in the agilization of Intel. He is co-author of *Change-Based Test Management: Improving the Software Validation Process* (ISBN: 0971786127), and he has spoken at a number of conferences worldwide.

Copyright Ray Arell 2009

Published at the Pacific Northwest Software Quality Conference 2009

Introduction

Right off the get go, I have to admit that Agile project management three years ago ran contrary to almost every development process fiber in my body. After all, I participated and drove part of the creation of the waterfall product life cycle (PLC) for Intel platform software. I also used the Intel PLC within my own development teams, and I linked to other teams that followed the methods as well. So what would make a person so vested in the waterfall lifecycle rebel the way I have? Simple, the Waterfall model is flawed.



In the Waterfall model, Requirements are expected to be locked down 6+ months prior to the completion of the project. Doing so, as you can see from the figure above, creates an environment that is very resistant to change once the project is under development. This smothers innovation and makes it very hard to quickly respond to customer needs, and it creates a negatively charged work environment and high customer frustration.

Delivery cadence is also a major flaw. Requirements are never perfect no matter how well you think they are documented. Not giving customers a chance to have frequent review means you have a strong chance of missing the customer expectations due to miscommunication. That could mean five months of rework because of how far the product has been built out. This leads to cost overruns, low employee satisfaction, and customer frustration.

I spent years trying to overcome these problems, but I got to the point that I knew the model was flawed and we needed to try something new. That is when I started to dig into Agile methods and decided to use the Scrum project management methodology.

It is important to note that the scope of this paper only represents a handful of teams at Intel. Our corporation has hundreds of teams using a variety of different development methodologies, and this paper is not to imply that Intel as a whole is moving to an Agile development model. Though, like the rest

of the industry, Intel is starting to have a significant groundswell of new teams adopting Scrum and other Agile methods within the corporation.

What Sold Me on Agile

The primary selling point for Agile was its core focus on delivering high customer value with every release. Plus, I liked that it put together a work culture that was focused on teamwork, just enough process to get stuff done, frequent customer feedback, and, most of all, accepting the reality that requirements can never be fully determined at the start of the program.

Picking the Project Framework

Selecting the project framework was probably one of the most critical parts of going to the Agile development model. The Scrum model seemed to work the best. The framework was well structured and allowed modifications, it addressed building customer value and the work flow, and it would drive a culture of transparency and openness around the project. I also liked that it was not prescriptive on how the work got done. It was flexible enough to drop in any appropriate methods or tools.

Moving from the Waterfall to Scrum

Moving from waterfall to Scrum is not an easy task. Mostly, because it is a major paradigm shift for the people involved. A number of people thought the best way was to evolve over time to the Scrum methodology, but it is hard to see an effective path to doing this when you're dealing with a small co-located team. So we switched to a Scrum framework as soon as we had the team trained and a Scrum Master certified.

Effect on the Team

Within the first several product iterations, the effect on the team was clear. We had issues with old school behaviors around the role of management, people using job titles to avoid taking on tasks, problems with self-management, and the perception that the standup meetings were a form of micromanagement.

The role of the manager in the Scrum process required a fair amount of un-learning for me. I needed to shift from somebody who assigned tasks to a person coaching and removing roadblocks. I needed to be the one who oiled the gears between the development team and the customer, not the guy asking, "Did you get that done?"



People's job titles also drove a destructive behavior in the team. For example, people with the title of architect would not pick up any task they felt was not their responsibility even if it was in their skill set, and this broke down team communications and trust. This led me to make the decision to abolish job titles, and I reinforced that people needed to embrace the concept that an Agile team member will take on any task to insure we meet our high value release. I knew making that decision ran the risk of losing a few people, and, in fact, a few people left to join a waterfall team. It was unfortunate to lose their skills, but, in retrospect, it enabled our adoption to move faster and teams to work with a greater degree of teamwork.

The perception of micromanagement was something that I did not expect to come up. I thought that people meeting every morning would promote an open work environment and a higher level of cooperation between team members. It did, mostly, but a few in the team felt that being asked daily about what they did, what they were setting out to do, and if they had any blocking issues was micromanaging them. Now the interesting thing was this was not really the issue. Digging in deeper, this had to do with the fact that what they did yesterday was very transparent to the rest of their peers. You now had daily accountability to your team. The people were mad at the process because it did not allow them to slack off.

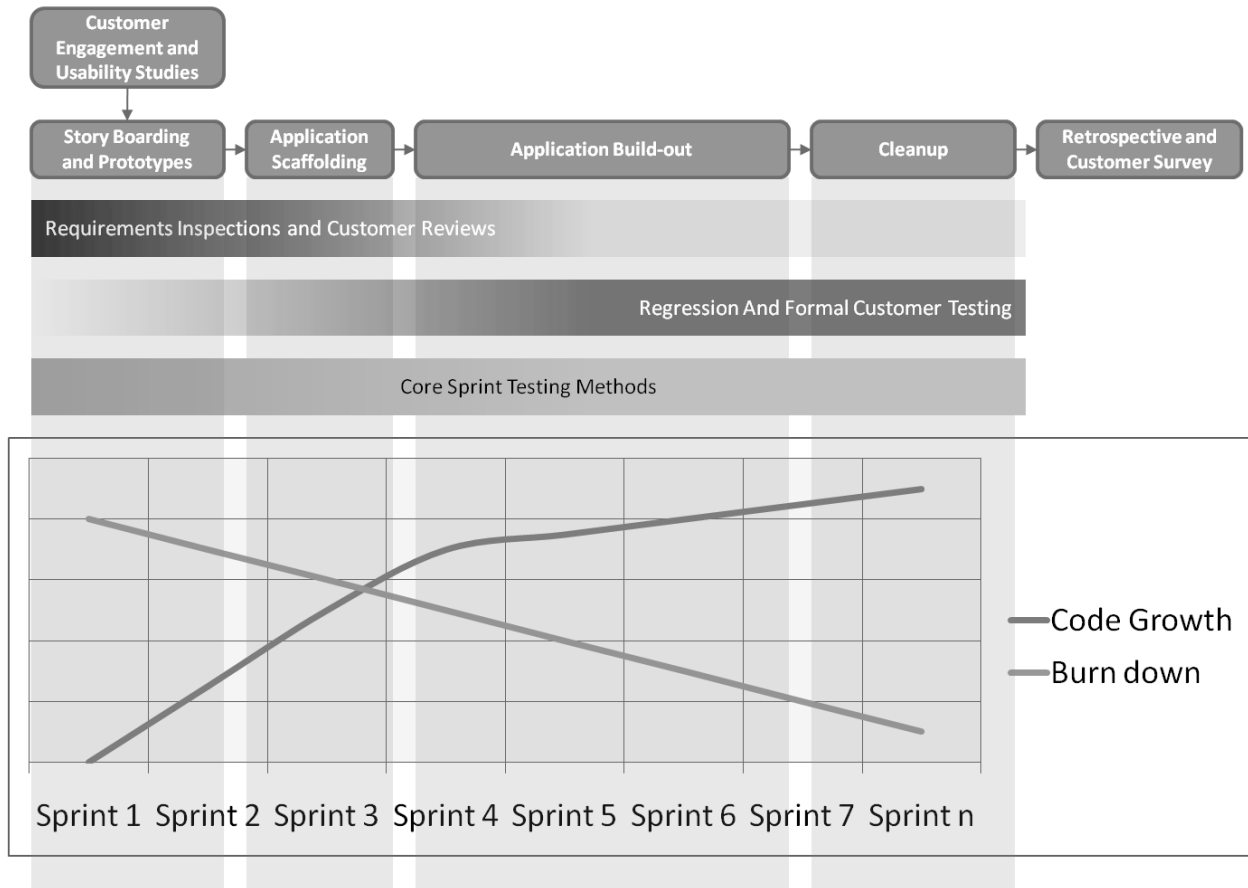
Specific Effect on Testing Professionals

Intel has built up a very strong validation capability built around the Waterfall PLC. In fact, a fair number of validation teams inside of Intel have developed their own waterfall model that runs parallel to the primary development PLC. For a test professional to transition into an Agile environment is difficult. Existing test strategies, plans, test development, automation, and other processes do not take into account the potential product requirement changes. Furthermore, testing teams inside of Intel tended to be segregated from the product developers. So working integrated with the developers in the Scrum process was very awkward and stressful at the start.

Scrum Software Validation Strategies

The most important document that is created for any project is the test/validation strategy document. This holds true if you're using Scrum or the Waterfall lifecycle for your product development. For people who are coming from the waterfall environment, it is very important to note that you need to approach the strategy with a clean slate. This is primarily due to the fact that the Scrum project framework puts two major constraints on testing: 1) Products are delivered on a fixed cadence, and cannot be pushed out. 2) All features need to be working and meet the acceptance criteria. In other words, no features are shipped to the customer if it is not tested, repaired, and retested. Let's look at one possible strategy.

The illustration below shows the degree and type of testing deployed throughout the development lifecycle. You'll note a more aggressive posture on requirements inspection is done early on in the project when the application is in its early stages of development. Once the software is built out, a heavier emphasis is put on regression and customer testing. Atomic level feature testing is performed with each of the sprint story deliverables, and this occurs consistently throughout the life of the development. In the early stage of product development, a usability study is conducted in order to build a pseudo-representation of the end-user. This persona, which we will explore later in this paper, enables the development team to focus on the high value needs of the customer.



Strategy: Test Scheduling

Within the Scrum framework, user stories are the mechanism for capturing product requirements. They describe the functionality with respect to the value to the customer and they contain a high-level description of the feature, evolving details, and the acceptance criteria that will be used to judge whether or not development has been completed.

One of the most profound realizations in the Scrum methodology is that putting too much detail into an ambiguous definition of a feature is a waste of time. Instead, details are deferred until the maximum amount of information is available. And, in most cases, the product development itself helps to fill in the gaps. So, as nerve-racking as it seems, test planning needs to be distributed within each development sprint.

At the beginning of each sprint, the test engineer will establish a test strategy on how to test each of the atomic features being delivered in the iteration, and will define what level of regression testing will be performed on the release candidate prior to distribution to the customer. All necessary infrastructure changes, like validation tool modifications, become a part of the development tasks that are tracked and performed within the sprint. Test case development for each of the features is also created within the sprint and ran within the sprint.

Strategy: Requirement Inspections and Regression Testing

You'll be happy to hear that requirement inspection and regression testing don't change that much from what you would use in the waterfall model, but it is important to understand how the key constraints, as talked about above, factor into what flavor of methodology that you employ.

Requirements inspection of the user stories takes place at the beginning of each sprint, and is primarily focused on answering one critical question, “Is the user story testable?” If it does not pass that scrutiny, then it should not be accepted into the development cycle unless there's a clear path to get that information. After all, if you can't test it, then how are you going to build it?

Regression strategies should be based on a change-based/risk-based methodology to ensure that your product quality has not regressed. Furthermore, your regression needs to be fast. If you break down a two-week development cycle, then you will see that a continuous integration and regression methodology will be needed to address the very short amount of time that you have to perform testing prior to shipping the product. The nightly regressions should be focused more on a smoke test of the product, and the targeted regression used in the days prior to shipping needs to cover a wide breadth and selected depth of testing to ensure high product quality. One key note on test automation, automate only what is necessary. Stick with automating the highest value test cases, and structure your automation infrastructure to allow for easy modifications and changes.

Strategy: Focusing your Effort with Customer Personas

One of the most daunting parts of establishing a good test strategy on a feature rich product is learning how the customer is going to interact with your application. In the Agile world of building high customer value with each release, you need to set a test strategy that will target the behaviors, skill, work environment, and expectations of your users. If you have only several customers, this is not too difficult to understand. But what if you have 30,000+ users of your product? Well that is where the customer persona comes into play.

A persona is a fictional person that is created to represent an amalgam of the demographics of the target users of your product. It captures all the critical characteristics of your users. Typically the persona is synthesized from a collection of interviews of real target users. Depending on the sample size of people interviewed, you may end up with more than one persona, which can be used to further focus your test effort. The theory is that if you meet the expectations of the personas, then you meet the needs of most of your users.

Example Persona

To help illustrate this, let me talk about the fictional character who we will call Bill Johnson.



Bill is an information technology professional with more than 15 years experience. He spends his day troubleshooting end user problems on a help line in a company of 8,000 employees, and he is also responsible for making recommendations on new technologies that help optimize cost. He is a proficient user of most office and server tools. He is a very detailed reviewer of products, and he tends to focus on compatibility of applications within his existing computer network. His pet peeves with any product are complex, hard to learn interfaces; simple errors in documentation and help files; basic features working intermittently under high network loads; and basic features that do not work the way they were described on sales brochures. Any of these conditions will prompt him to give a negative recommendation on a product. Because his recommendations have corporate-wide impact, he will vet all features first in his

lab, progress to a limited pilot, and finally do a full deployment on his corporate network. He seldom uses advanced features of the product. If he does, he will use samples provided by the vendor....

The persona continues to go on talking about the list of applications that he uses by name, what he likes and dislikes about competitive products, types of defects that are unacceptable, etc.

At this point you should be seeing the value of the persona. The fictional character, Bill, puts a face to a group of users of a product. If we focus on satisfying him, then we are on our way to meeting the expectations of a large group of people.

Applying the Persona to Your Test Strategy

The main addition to your strategy will be a new section that gives a brief overview of your user personas, the location of the detailed documents, and how the personas were used to set your approach. This level sets the customer-relevant information that will be interwoven throughout the remainder of your document.

Now let's use our fictional character, Bill Johnson, to help us to choose the testing methods we might want to use. We know, based on Bill's profile, that his testing is very exploratory in nature. He starts with new features, and later he will check a new product's ability to work alongside other currently deployed products. Based on this information, it may be a good idea to add Session-Based testing to our strategy and to use his persona in the creation and priority setting of each of the testing sessions. As you can see, the benefit we have gained is that part of our testing is going to be similar to Bill's. This is a vital step to finding defects in your lab versus his. It is one example of focusing the methods used, and you should continue to pick other methods and processes that will continue to drive a positive usage experience.

Bill also helps us to analyze gaps and limitations to our strategy, to convey the possible impact to our customer if a defect escapes, and to form a mitigation plan if we run the risk of a major test hole. We know from the profile that Bill is going to test the product in three environments. The largest is a network with 8,000 nodes. With no mitigation, we are at risk of Bill finding defects when he enters the third stage of his evaluation. Also, we know that those issues are not going to be found right away. The timing of Bill's testing is not called out in the persona, but we can make a fair assumption that those issues may come weeks to months later after the product has been deployed adding to the risk. One other possible risk could be that you have more than one user persona, and you may not be able to test to more than Bill's expectations. With the personas of those additional users, you are able to articulate possible market impact and risk of not being able to test to that demographic. In both cases you are making data driven tradeoffs and decisions based on real customer impact.

At this point you should have a good overview of what a customer persona is and how it fits into your test strategy. Using this method will make having to create a well formed test strategy less daunting, and it will focus your work on protecting a positive user experience. Done right, you can expect a big "thank you" from all the Bill Johnson's in the world.

Conclusion

As talked about earlier in this paper, I want to make sure that no one gets the impression that Intel has converted to a 100% Scrum process. We still have a lot of waterfall teams working within the company, and those teams deliver products with respectable quality. But, from my perspective, I do believe that the Scrum process and other Agile methods are a far more effective way of meeting customer needs because they are centered on delivering high value to the customer with every release.

I hope that this guidance on how to structure a testing strategy for Scrum gives you some insight into what you might be in for when you transition your team to the Agile Scrum development model.

References

Pettichord, Kaner, Bach, *Lessons Learned in Software Testing*, on-line
Various Authors, Exploratory Testing, Wikipedia
Various Authors, Test Strategy, Wikipedia
Various Authors, *Scrum (development)*, Wikipedia
Various Authors, *Session-based testing*, Wikipedia
The Scrum Alliance, <http://www.Scrumalliance.org/>
Ray Arell, *Change-Based Test Management*, (ISBN: 0971786127)
James Bach, *Heuristic Risk-Based Testing*, STQE 11/99
James Bach, *Risk and Requirements-Based Testing*, Computer, June 1999
Ingrid Ottevanger, *A Risk-Based Test Strategy*, StarEast 2000
Bret Pettichord, *The role of information in Risk Based testing*, StarEast 2001
Erik Petersen, *Smarter Testing with the 80:20 Rule*, StarWest 2002
Anne Campbell, *Using Risk Analysis in Testing*, StarEast 2000
Paul Gerrard, *Risk-Based E-Business Testing*, System Evolutif
Gregory T Daich, *Defining a Software Testing Strategy*
Jim Highsmith, *Agile Project Management*
Ruku Tekchandani, *Building a Effective Test Strategy*, Intel SQE

Managing Software Debt

Continued Delivery of High Values as Systems Age

Chris Sterling
Principal Consultant
SolutionsIQ
Email: csterling@solutionsiq.com

Author's Bio

Chris Sterling is an Agile Coach, Certified Scrum Trainer, and Technology Consultant for SolutionsIQ. He has been involved in many diverse projects and organizations and has extensive experience with bleeding edge and established technology solutions. He has been a coordinator of multiple Puget Sound area groups including International Association of Software Architects (IASA), Seattle Scrum Users Group, and most recently the Beyond Agile group. He has been a speaker at many conferences and group meetings including Agile 2007 & 2008, SD West, Scrum Gathering, and others. In his consulting and speaking engagements, Chris brings his real world experience and deep passion for software development enabling others to grasp the points and take away something of value. Chris has also contributed to and created multiple open source projects. He is currently teaching the "Advanced Topics in Agile Software Development" class at the University of Washington Agile Developer Certificate extension program and writing a book with publisher Addison-Wesley on software architecture.

Abstract

Many software developers have to deal with legacy code at some point during their careers. Seemingly simple changes are turned into frustrating endeavors. Code that is hard to read and unnecessarily complex. Test scripts and requirements are lacking, and at the same time are out of sync with the existing system. The build is cryptic, minimally sufficient, and difficult to successfully configure and execute. It is almost impossible to find the proper place to make a requested change without breaking unexpected portions of the application. The people who originally worked on the application are long gone.

How did the software get like this? It is almost certain the people who developed this application did not intend to create such a mess. The following article will explore the multitude of factors involved in the development of software with debt.

What Contributes to Software Debt?

Software debt accumulates when focus remains on immediate completion while neglecting changeability of the system over time. The accumulation of debt does not impact software delivery immediately, and may even create a sense of increased feature delivery. Business responds well to the pace of delivered functionality and the illusion of earlier returns on investment. Team members may complain about the quality of delivered functionality while debt is accumulating, but do not force the issue due to enthusiastic acceptance and false expectations they have set with the business. Debt usually shows itself when the team works on stabilizing the software functionality later in the release cycle. Integration, testing, and bug fixing is unpredictable and does not get resolved adequately before the release.

The following sources constitute what I call software debt:

- Technical Debt[1]: those things that you choose not to do now and will impede future development if left undone
- Quality Debt: diminishing ability to verify functional and technical quality of entire system
- Configuration Management Debt: integration and release management become more risky, complex, and error-prone
- Design Debt: cost of adding average sized features is increasing to more than writing from scratch
- Platform Experience Debt: availability and cost of people to work on system features are becoming limited

Software Debt Creeps In

Feature Area	Component 1	Component 2	Component 3
F1			
F2			
F3			
F4			
F5			

Figure 1: This figure shows the amount of software debt in a fictional piece of software's first release. The white cells represent well-crafted areas of functionality contained in a specific component of the software's architecture. The dark gray areas represent functionality that is starting to show signs of software debt.

Figure 1 displays a system that has minimal amount of software debt accrued. A few low priority defects have been logged against the system and the build process may involve some manual configuration. The debt is not enough to significantly prolong implementation of upcoming features.

Business owners expect a sustained pace of feature development and the team attempts to combine both features and bugs into daily activities, which accelerates the accrual of debt. Software debt is accruing faster than it is being removed. This may become apparent with an increase in the number of issues logged against the system.

As a system ages, small increments of software debt are allowed to stay so the team can sustain their velocity of feature delivery. The team may be complaining about insufficient time to fix defects.

At this point, delivery slows down noticeably. The team asks for more resources to maintain their delivery momentum, which will increase the costs of delivery without increasing the value delivered. Return on investment (ROI) is affected negatively, and management attempts to minimize this by not adding as many resources as the team asks for, if any.

Even if business owners covered the costs of extra resources, it would only reduce the rate of debt accrual and not the overall software debt in the system. Feature development by the team produced artifacts, code, and tests that complicate software debt removal. The cost of fixing the software debt increases exponentially as the system ages and the code-base grows.

Feature Area	Component 1	Component 2	Component 3	Component 4	Component 5
F1					
F2					
F3					
F4					
F5					
F6					
F7					
F8					
F9					
F10					
F11					
F12					

Figure 2: This figure shows software debt has continued to creep accrue in all functional areas and architectural components. There are less areas of functionality that are not affected by software debt. The darker the cell is, the more debt that area of the software has accrued.

Software debt in the system continues to accrue over time, as shown in figure 2. At this point, new feature implementation is affected significantly. Business owners may start to reduce feature development and put the system into “maintenance” mode. These systems usually stay in use until business users complain that the system no longer meets their needs.

Managing Software Debt

There are no magic potions for managing software debt. Software can accrue debt through unforeseen circumstances and shortsighted planning. There are some basic principles that help minimize software debt over the lifespan of the product:

- Maintain one list of work
- Emphasize quality
- Evolve tools and infrastructure continually
- Always improve system design
- Share knowledge across the organization
- And most importantly, hire the right people to work on your software!

Maintain One List of Work

One certain way to increase software debt is to have multiple lists of work. Clear direction is difficult to maintain with separate lists of defects, desired features, and technical infrastructure enhancements. Which list should a team member choose from? If the bug tracker includes high priority bugs, it seems like an obvious choice. However, influential stakeholders want new features so they can show progress to their management and customers. Also, if organizations don't enhance their technical infrastructure, future software delivery will be affected.

Deployed software considered valuable to its users is a business asset, and modifications to a business asset should be driven from business needs. Bugs, features, and infrastructure desires for software should be prioritized together on one list. Focus on one prioritized list of work will minimize confusion on direction of product and context switching of team members.

Emphasize Quality

An emphasis on quality is not only the prevention, detection, and fixing of defects. It also includes the ability of software to incorporate change as it ages at all levels. An example is the ability of a Web application to scale. Added traffic to the web site makes performance sluggish, and becomes a high priority feature request. Failed attempts to scale the application result in a realization that the system's design is insufficient to meet the new request. Inability of the application to adapt to new needs may hinder future plans.

Evolve Tools and Infrastructure Continually

Ignoring the potential for incremental improvements in existing software assets leads to the assets becoming liabilities. Maintenance efforts in most organizations lack budget and necessary attention. The International Organization for Standardization (ISO) standardizes on four basic categories of software maintenance in ISO/IEC 14764[2]:

- Corrective maintenance - Reactive modification of a software product performed after delivery to correct discovered problems
- Adaptive maintenance - Modification of a software product performed after delivery to keep a software product usable in a changed or changing environment

- Perfective maintenance - Modification of a software product after delivery to improve performance or maintainability
- Preventive maintenance - Modification of a software product after delivery to detect and correct latent faults in the software product before they become effective faults

Most maintenance efforts seek to prolong the life of the system rather than increase its maintainability. Maintenance efforts tend to be reactive to end user requests while business evolves and the technology decays.

To prevent this, attention must be given to all four categories of software maintenance. Budgets for software projects frequently ignore costs for adaptive, perfective, and preventive maintenance. Understanding that corrective maintenance is only part of the full maintenance picture can help an organization manage their software assets over its lifespan.

Improve System Design Always

Manage visibility of system design issues with the entire team. Create a common etiquette regarding modification of system design attributes. Support the survival of good system design through supportive mentoring, proactive system evolution thinking, and listening to team member ideas. In the end, a whole team being thoughtful of system design issues throughout development will be more effective than an individual driving it top down.

Share Knowledge Across the Organization

On some software systems there is a single person in the organization who owned development for 5 years or more. Some of these developers may find opportunities to join other companies or are getting close to retirement. The amount of risk these organizations bear due to lack of sharing knowledge on these systems is substantial.

Although that situation may be an extreme case of knowledge silos, a more prevalent occurrence in IT organizations is specialization. Many specialized roles have emerged in the software industry for skills such as usability, data management, and configuration management. The people in these roles are referred to as “shared resources” because they use their specialized skills with multiple teams.

Agile software development teams inherit team members with specialized roles, which initially is a hindrance to the team’s self-organization around the work priorities. Teams who adhere to agile software development values and principles begin to share specialized knowledge across the team, which allows teams to be more flexible in developing software based on priorities set by business. Sharing knowledge also reduces the risk of critical work stoppage from unavailable team members who are temporarily on leave.

Hire the Right People!

It is important to have the team involved in the hiring process for potential team members.

Teams will provide the most relevant skills they are looking for, thus, allowing them to review and edit the job description is essential. Traditional interview sessions that smother candidates with difficult questions are insufficient for determining if a candidate will be a great fit.

Augmenting the interview questions with a process for working with the candidate during a 1 to 2 hour session involving multiple team members in a real-world situation adds significant value to the interview process. Before hiring a candidate, team members should be unanimous in the decision. This will increase the rate of success for incorporation of a new team member since the team is accepting of their addition.

Another significant hiring focus for organizations and teams is placing more emphasis on soft skills than technical expertise. I am not advocating ignoring technical experience. However, it is critical in an agile software development organization or team to have people who can collaborate and communicate effectively. Soft skills are more difficult to learn than most technical skills. Look for people who have alignment with the hiring team's values and culture.

In Summary

As systems age they can become more difficult to work with. Software assets become liabilities when software debt creeps into systems through technical debt, quality debt, configuration management debt, design debt, and platform experience debt.

Applying the six principles in this article will lead to small changes that over time will add up to significant positive change for teams and organizations. The goal of managing software debt is to optimize the value of software assets for our business and increase the satisfaction of our customers in the resulting software they use.

References

1. Ward Cunningham - "Technical Debt" - <http://c2.com/cgi/wiki?TechnicalDebt>
2. "ISO/IEC 14764: Software Engineering -- Software Life Cycle Processes -- Maintenance" - International Organization for Standardization (ISO), revised 2006

My Experience at Adopting an Agile Software Development Approach

John Bartholomew
Nethra Imaging, Inc., Beaverton, OR
bart@nethra.us.com

Abstract

This paper presents an overview of the software development process used at Ambric Inc., a fabless semiconductor startup company in Beaverton OR, from roughly 2006 to 2008. During that time, an agile development process was initiated and refined over the course of several major software releases, yielding significant improvements in both product quality and in reliability of release delivery dates. An analysis of the direct benefits of this approach and additional insights gained are included.

Our team of ten software tool developers and four quality assurance engineers produced six major releases in roughly two years. In addition, a smaller application development team and their associated quality assurance engineers also used an agile approach in delivering their releases. Only the tool team's efforts will be covered in this paper. For both teams, the resulting products supported Ambric's massively parallel processor array (MPPA) chip, which allowed our customers to create and debug designs to program the chip's 344 processors and on-chip memory. Customer and internally produced applications were in the areas of video encode/decode, encryption/decryption, and image and signal processing in several domains.

Biography

John has worked in the EDA (Electronic Design Automation) and semiconductor software tools industries for over 20 years in the Portland OR area, working primarily on simulation and co-simulation toolsets. He has also served as an adjunct faculty member at the Oregon Institute of Technology, and Portland Community College.

While the company described in this paper, Ambric, closed in November 2008, its intellectual property was purchased by Nethra Imaging of Santa Clara CA in early 2009. John and several former Ambric employees are now continuing to develop the Ambric multi-processor array chip technology and software toolset at Nethra. John is currently the Senior Software Architect for the the Ambric tool set at Nethra Imaging.

John has an M.S and B.S in Electrical Engineering/Computer Science from the Massachusetts Institute of Technology.

The Scrum Methodology

The development model we adopted was based heavily on the agile programming practice known as “Scrum”. [1] Scrum can be considered a lighter weight alternative to the more radical XP (Extreme Programming) approach. [2] Where XP focuses more on the details of programming aspects of the development process, Scrum is concerned with the methodology and framework of that process – i.e., how your team effort is organized and executed, not the tactics of the daily work flow.

A key principle of Scrum is its recognition that during a project the customers can change their minds about what they want and need and that unpredicted challenges cannot be easily addressed in a traditional predictive or planned manner. As such, Scrum adopts an empirical approach—accepting that the problem cannot be fully understood or defined, focusing instead on maximizing the team's ability to deliver quickly and respond to emerging requirements. [3]

In the Scrum methodology, a *Product Owner* ensures that the development team has the right input from the company's business perspective. For us, this meant that management (including sales/marketing) negotiated the general content of a small number of releases into the future, with software team leads quickly estimating the effort required to design, implement and test new features. The result was a short prioritized list of planned new features per quarterly release, based on actual and expected customer need. This list is referred to as the *product backlog*. These new features were not set in stone at this point, but served as the prioritized initial road map for the development team when starting a new release cycle. If the priority of a scheduled feature changes during the course of the release, the Product Owner must bring this fact to the attention of the team. The team will then alter their development plans accordingly. This flexibility is one of the principle benefits of an agile approach; the iteration planning process must be thorough but also lightweight enough to easily accommodate such in-flight changes. The role of Product Owner was split between our VP of Software Engineering and our marketing Product Manager.

In Scrum, the basic unit of team effort is called an *iteration* or a *sprint*. Each iteration is a few weeks in length, often in the two to four week range. One of the main goals of Scrum is that each iteration adds well tested increments to the overall product functionality, as shown in Figure 1, resulting in a new shippable product release at the end of each iteration.

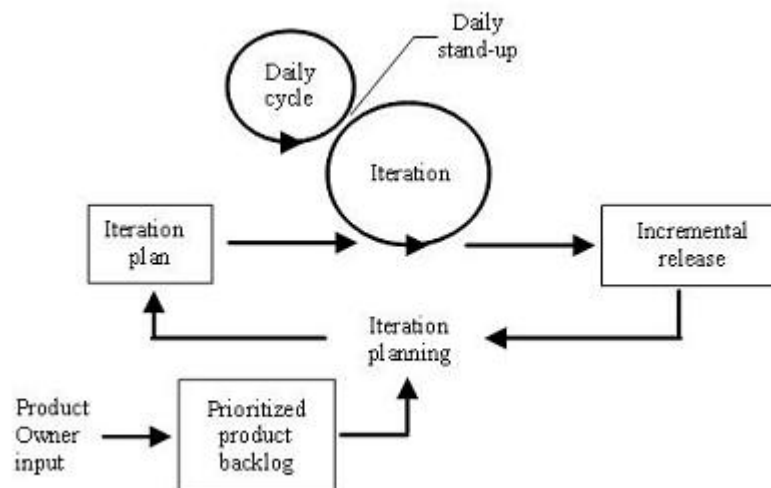


Fig. 1 The Scrum process [1]

In order for our product to have major releases available once each quarter, a release typically included six iterations, with the final iteration being three weeks long to allow completion of more extensive release testing. At the start of the first iteration, teams would work through the design of all new features in the release in detail with a task breakdown in order to verify that implementation and testing of these features could be completed within the scope of that release. If not, the release contents had to be renegotiated with the Product Owner or resources had to be reassigned as needed. Our software team manager filled the *Scrum Master* role, tracking progress towards team goals and performing any needed renegotiation or reassignment. This worked well within our team, due to the smaller size and lack of organizational hierarchy common in most startups. It may not be generally advisable, however, due to the possible conflicting roles of team manager (directing work, providing performance reviews) and Scrum Master (facilitating development but not directly assigning it).

A typical quarterly release structure for our product is shown in Table 1, and our development iteration's structure is shown in Table 2.

Iteration 1	Feature planning, design, initial documentation; development start
Iteration 2	Development/test
Iteration 3	Development /test
Iteration 4	Development /test
Iteration 5	Development /test
Iteration 6 [three weeks]	Release testing, final bug fixing (no new feature code), release packaging

Table 1. Typical Scrum release structure

	Monday	Tuesday	Wednesday	Thursday	Friday
Week One	Feature design; coordinate team interdependencies	Develop/test	Develop/test	Develop/test	Develop/test
Week Two	Develop/test	Develop/test	Develop/test	Qualified release build/test	Fix outstanding issues, if any

Table 2. Scrum iteration structure

Scrum-based methodologies often include a brief (15-20 minutes) daily team project status meeting, called a *stand-up meeting*. Based on several team members' prior experience, we chose to hold only two each week, held by individual teams (compiler, debugger, etc.) not the entire software team as a whole. The entire software team would gather only once per week for an overall project status review, as well as once at the start of a new iteration. This latter meeting would serve as a review of the prior iteration's results, as well as a kick-off for the new iteration about to start.

The Scrum Iteration Model

Early on, we decided that all development, including significant test development, would be performed on branches from the main code repository. In this way, both developer implementation and unit testing and QA team testing would be performed on any new feature prior to it being committed to the code repository main line. At first, several team members were unhappy with the decision to use branches; this disliked the extra overhead of branching and merging, and they had typically developed independently and checked directly into the database main line. As it turned out, each sub-team (2-4 members) working on a given feature could be isolated from other teams, yet coordinate their own work, via a common branch.

Branches were kept alive for as short a time as possible, in order to achieve a level of completion that could be tested as non-disruptive to the main line, and then be merged. Longer-lived branches would be periodically updated from the repository main line to ease merging later on.

As part of this practice, QA team members were tightly integrated with development teams, involved as early as reviewing initial documentation during new feature design. This enabled QA team members to more fully understand new features and to complete their test planning in coordination with their associated development team lead. QA team members also used their development team's branch for their test efforts, including running a full regression pass to test for integration failures. New code (features or significant bug fixes) could therefore be developed and more completely tested as a modular unit prior to being included in the code repository main line. This reinforced the Scrum goal of producing a shippable product at the end of each iteration.

If a new feature testing was not yet completed at the feature commitment cutoff point (iteration week 2, Wed. eve), then the responsible developer had to hold off merging any of their changes from their branch to the repository main line until that iteration's qualified build was tested and declared good. This practice tended to eliminate surprises due to the inclusion of buggy, partially implemented/partially tested code checked in to the repository at the last minute. Instead, the slightly-behind-schedule developer would finish the work on their branch over the next day or two. Once the current iteration qualified release was declared completed, the code repository main line was opened once again for committing new code. In the meantime, in any time remaining prior to the start of the next iteration, this lagging developer could start work on features scheduled for that next iteration. The only loss is that the feature originally scheduled to be completed in iteration N is now completed in iteration N+1, but without the quality risk of a hastily completed piece of functionality.

A very important aspect of the development flow was for the Scrum Master to monitor new feature progress during the first half of a release. If any particular team was falling behind schedule, that team's remaining unimplemented features for the current release would be renegotiated with the Product Owner. The Scrum Master can make this determination based on a feature of Scrum toolsets called a *burndown chart*. This is a graph showing the time estimate of the remaining work for the current iteration (the *iteration backlog*) versus the time elapsed. Ideally, this graph should show a convergence towards zero remaining work at the end of the iteration (or release). See Figure 2 for an example.

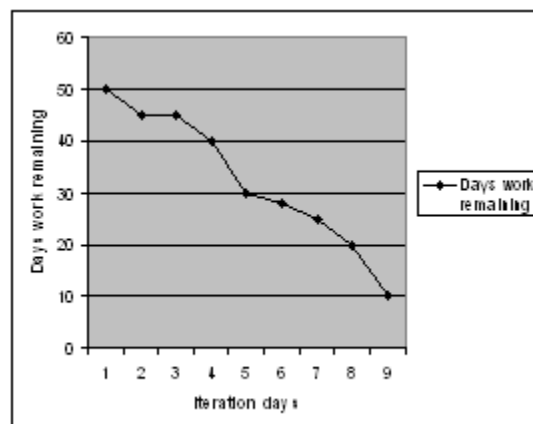


Fig 2. Iteration burndown chart

If one of these jeopardized features could be pushed into the next release, it would be. The idea here was that any customer need for such a feature could be addressed in the short term by an end-of-iteration qualified build early in the following release. (A customer would see only a few weeks' delay in the

delivery of a feature that had slipped, rather than a full quarterly release cycle.) As part of this re-planning process, it was important for the Product Owner to provide a properly prioritized feature list at the start of each release. For risk reduction, release critical features would not be planned for completion by the development team in the second half of a release cycle if at all possible.

An alternative approach to moving features out to a later release was to add an additional iteration(s) to the current release, effectively extending the release date. While the team responsible for the late features could complete their tasks, this had a side effect of delaying other developers. If they were done with their new features and critical bugs for the current release, there was an inherent risk in allowing these developers to continue to add new features or bug fixes into the current release in the final development iteration(s). These other developers could work on planning and implementation for the next release, but they had to work on a branch and not commit any new code to the repository main line until the current release was completed.

Testing

Nightly builds and regression tests were run to ensure that all code checked into the repository main line would build and pass all tests. Any large disruptions in test results were addressed immediately, with one or more developers setting aside current work to trace and implement the needed fix. Small changes to test results (for example, a few new failing tests out of our few thousand regression tests run each night) would be inspected to determine the root cause. A bug report would be filed to be fixed as appropriate within the current iteration (if small) or the current release (if more significant work was involved). In the worst case, the code changes causing the failure could be backed out, and the new feature fixed on a branch prior to being further tested and checked back into the repository when ready.

In addition to regular developer and QA testing, whole team “bug hunts” were included in each release, typically one part way through release, and one near the end. A bug hunt entailed a half-day testing effort where all team members read up on one or more new features (preferably not in their area) and attempted to expose problems. Typical strategies often included corner case tests, invalid or null parameters to APIs/UI dialogs/language constructs, simulating “new user mistakes”, capacity testing, and so on.

The QA team manager held a weekly bug scrub meeting with the team leads to triage recent new bug reports. This meeting served to identify critical bugs which had to be addressed immediately, redirect inappropriately filed reports, detect duplicate bug reports and generally acted as an early warning system for overall release quality. Towards the end of a release cycle, this meeting would be held more frequently and focused only on release critical bugs. All other non-critical bugs were reassigned to be fixed in a future release.

Benefits of this Approach

This model provided the following benefits over our prior, less formalized software development approach:

Thorough testing of new features (including significant bug fix tasks) on a branch by both developers and QA team members significantly reduced the introduction of bugs to the main code repository. This was a significant contributor to the quality level of each iteration's qualified release build. Tight integration of the development and QA teams was required to achieve this, however; development team unit testing alone was not sufficient.

Qualified release builds were able to be produced roughly every two weeks, in addition to a full release every three months. The qualified release builds could be delivered to a customer as an interim release

for critical bug fixes or early access to new features. These frequent small releases provided a sense of consistently accruing new features in the product; our former longer release cycles tended to engender feelings of “there’s still so much to get done before the next release...”.

As we incrementally improved our development process, there was markedly less “death march” feeling to the end of a release (e.g., fixing a sea of critical issues, uncontrollable feature creep, etc.). This provided far more predictable release quality and release delivery date with far less stress and higher team morale.

Our last few releases were delivered with acceptable quality and all or nearly all initially planned features within one week of the originally planned quarterly release date. Hitting multiple three month release cycles within one week of their planned dates provided significant credibility of our approach to our executive management team. It also gave our sales team and field force confidence in our ability to meet promised deliveries.

Key Findings

Refine, refine, refine. It took several releases to refine our use of the iteration model to the point where team members were comfortable and productive. This included being better able to estimate not only a feature’s development time but allowing for the testing and associated initial bug fixing phase prior to merging new features into the code repository main line of development. As the development and delivery process improves, it builds trust with your field and sales force as well.

Less isn’t always more. We chose a two week iteration length, primarily due to suggestions from our Scrum tool set provider at the time. Some team members, however, felt that a longer iteration time (3 to 4 weeks) might have provided more development time with less test/release overhead. In addition, with a shorter iteration time, new feature development was nearly always performed on a branch, to avoid having partially completed new feature code checked in to the repository main line at the end of the iteration. A longer iteration time would allow developers to use the code repository main line for much of an iteration, only resorting to branches for features started near the end of an iteration.

Be willing to bend the rules – a little. Having the patience and fortitude to declare an iteration’s qualified build as incomplete due to a discovered critical issue was hard at first, as it delayed the completion of the current iteration’s release – with the harsh side effect of closing the code repository main line to code changes. Extending the current iteration by a day or two to solve such issues, and then completing the qualified build, kept the confidence in our end of iteration qualified builds high and new feature reliability high as well. Having an adaptable process rather than a iron-clad one kept the team morale up.

Managing release quality takes time, and you must budget for it. We were “our own worst critics” - bug hunts would often turn up many small, corner case bugs or integration issues that customers likely would not encounter. Prioritizing these bugs (fix in current release vs. future release) was time consuming but necessary. We realized we simply couldn’t both fix all of the issues we’d turned up and deliver all needed new features in the current release cycle. As our product was still early in its life cycle, and many of our customers new to our product’s tool set, we favored providing new features over fixing every bug that our team had found internally. To be safe, our internally discovered, lower priority bugs were often left on the fix-in-current-release list until bug scrub meetings near the end of the release, when they were finally either fixed if time permitted or reassigned to be fixed in a future release. The choice of which bugs to fix and which to postpone was best made with the advice of one or more customer advocates, such as a field applications engineer.

Which comes first, the bug fix or the new feature? Managing these reassigned bugs was important in order to avoid a steadily increasing backlog of open bugs. Eventually, we employed team reviews of open bugs during release iteration 1 and included the needed bug fixes in the current release development task

list. Including only new feature development time in a release cycle led only to end-of-release bug fixing frenzies in order to meet our quality goals; allocating time to address the open bug list led to more reliable quality, but at the cost of pushing lower priority new features out into the future.

Verify quality early and often. Our first bug hunts were scheduled towards the end of each release cycle, and would produce a significant number of new bug reports. This tended to disrupt the end of release cycle, both in terms of quality metrics and development flow. We decided to split this one large bug hunt into two or more smaller ones, spread throughout the release cycle, focused on newly completed features. This spread the new feature bug fixing effort throughout the release cycle instead of loading up this effort at the back end.

When is “fixed” really fixed? During one of our first releases, we were far behind on bug verification as the release date approached. I volunteered to verify over 150 “fixed” bugs – and discovered that 1 in 15 was not, in fact, fixed. Some had been partially addressed, and some had regressed since the initial work was done. The longer you delay the verification of a bug fix, the more unexpected bug rework you may encounter at the end of a release. There’s a downside to early verification, however – code changes later in the release cycle may undo a prior fix. To prevent this, regression tests must be created for all non-trivial bug fixes and included in regularly run regression test suites to catch any such regressive behavior.

Stack the feature deck for success. At the start of a release cycle, we would make sure we’d scheduled all of the critical release features to be completed by the end of the third of the five development iterations. Any new features beyond that point – those scheduled in the fourth and fifth iterations – were considered candidates to be bumped out to the next release. This was all agreed to in advance between the team and the Product Owner. If the completion of work on high priority features was delayed, this prior ordering arrangement made it far easier to complete those higher priority features a little late and sacrifice a low priority release feature.

The team benefits as well as the product. The qualified build model really did allow us to have a new release every two to three weeks of reasonable quality for use by the field engineering team, or to act as an “alpha level” interim release to a customer. This boosted team morale as we weren’t constantly struggling with product quality concerns and could feel a constant growth in product features over time. This effect has been noted by other adopters of agile processes as well [4]. Team-wide bug hunts had a positive side-effect of allowing developers to learn about portions of the overall product that they didn’t directly work on. They also turned up incorrect/incomplete documentation issues, as testers worked from the new feature documentation available at the time.

A good workman doesn’t blame his tools – but does prefer good ones. The Rally Software tools [5] helped us formalize our development model as described above. We initially used another scrum-based tool, but with little success. The tool wasn’t really the issue in this case, however, it was our team adhering to and refining the iteration process model, which the Rally tool set supported very well. Having a good bug tracking tool was important, as well, especially towards the end of a release cycle when critical bug reviews were held daily. Here, Bugzilla fulfilled our needs quite well.

Conclusions

Our product was used in the field for demonstrations and delivered to many customers for evaluation, as well as a smaller number for actual use in their own product development. Overall, a very small number of bugs were ever reported by customers; our process was successful in catching nearly all potentially customer-visible issues in house.

Adopting the iteration development model described above took several releases, or the better part of a year, to evolve to the point where the team was both comfortable with it and adept at using it. The idea of pushing a feature out of a release and into the following release was difficult at first for management to accept; the ability to quickly provide that feature in an interim qualified build from an early iteration of that

next release at a delay of only a few weeks never turned out to present an obstacle for a customer, however.

Overall, this development model helped our software team to consistently deliver a tool set that was higher in quality and with far greater reliability of the delivery date and release features than was possible with our prior development process.

References

[1] K. Schwaber, M. Beedle, *Agile Software Development with Scrum*, Prentice-Hall, 2001.

[2] K. Beck, C. Andres, *Extreme Programming Explained*, Addison-Wesley, 2004.

[3] Wikipedia entry for the Scrum development process at [http://en.wikipedia.org/wiki/Scrum_\(development\)](http://en.wikipedia.org/wiki/Scrum_(development))

[4] S. Maguire, *Debugging the Development Process*, Microsoft Press, 1994.

[5] The Rally Software toolset at www.rallydev.com.

Improving Your Quality Process: A Practical Example

John Balza

Software Quality Consultant and Instructor

johnjbalza@comcast.net

Biographical Sketch

John Balza currently is a software quality consultant and teacher working with software companies to improve their software quality using inspections, defect analysis and metrics. John has presented at numerous software quality conferences since 2003.

John was the Quality and Productivity Manager at Hewlett-Packard in Fort Collins, Colorado. John was responsible for overall product quality for an organization of 1500 engineers in six geographic areas developing HP's version of UNIX, HP-UX. In this role, John reduced customer defects by over 80% and improved productivity by 30%. These changes were made by gaining management sponsorship for key process improvements and then facilitating teams to make these improvements. This included creating management metrics to track quality throughout the lifecycle, using inspections and defect analysis throughout the life cycle, applying agile methods to large complex projects, and assisting the lab organizations to achieve level 2 and 3 of the Capability Maturity Model. Prior to this assignment, John had managed over 50 software projects at several levels of management.

Abstract

Organizations are often faced with the problem of improving their quality process and justifying the investments for inspections and improved testing. This paper will discuss how Hewlett-Packard developed an improved quality process by:

- a) Understanding the types of defects that were escaping from the product,
- b) Using a model for the costs of finding and fixing defects to select various quality assurance techniques, and then
- c) Developing a quality process that included multiple inspections and test types.

Background

At the 2003 Pacific Northwest Software Quality Conference, I presented “Management Commitment to Quality Requires Measures”ⁱ that focused on the necessity to give management quality measures that were as available as schedule metrics. This paper explains the process side of that experience. It shares the business case we built for Hewlett-Packard’s (HP) management to improve our quality and the changes we made in our quality process.

The HP-UX product is Hewlett-Packard's version of the Unix operating system. It consists of approximately 18 million lines of code produced in a distributed organization of 1500 people in 7 worldwide locations. These 1500 people are organized into 13 different R&D labs reporting into several division managers. The product has a major release every 3 years or so, with quarterly updates in between. The size of these major upgrades had been doubling on each release. Each lab is responsible for the quality of their code and then one lab is responsible for the system testing. The HP-UX 11.00 version of our product was released with significant functionality additions to the previous release; in particular it was our first 64-bit operating system. Many customers began to use the operating system in mission critical applications where availability and reliability were very important. In these new environments, customers were demanding higher levels of quality than in our previous releases. In a survey conducted by the HP User Group, “developing higher quality software” was the 2nd highest issue in the poll.

Given the customer situation, management decided we needed to set an aggressive goal to improve our quality by 10X over 5 years – that is we would reduce the number of defects found by customers by a factor of 10 in those 5 years. This goal was chosen for two reasons – a) it was a familiar goal to the organization – in the 80’s a similar goal had been declared as a corporate goal, and b) 10X would get us to the same level of quality as a mainframe – so it was a good competitive goal.

In addition to the overall 10X goal, we decided we should have other goals that would help us achieve the long-term customer goal:

- Goal 1: subdivide the long-term goal. We chose an equivalent to 10X in 5 years, reducing customer defects by 2X every 18 months.
- Goal 2: reduce defects before they affect other engineers. We focused on reducing the defects before the code was checked-in. Check-in was the point where an engineer’s code was delivered to everyone in the project.
- Goal 3: Improve our system testing to further reduce the defects delivered to customers. It was general knowledge at the time that our system testing no

longer reflected our customers' environments – the typical customer used to be an engineer's desktop, but now it was large database applications in telecomm, finance and manufacturing organizations.

Target Improvements Using Defect Analysis

If we were really going to improve our testing, we needed to understand what improvements were needed. We asked every subsystem to conduct a defect analysis on the defects that were escaping to system test and to customers. Then we required a root cause analysis of the most frequent defect types with recommendations on what change was needed in our quality process to remove the root cause. Defect analysis has a key advantage over just imposing better quality processes. The engineering teams analyze their own data to understand what types of defects are escaping and then they recommend how to reduce these escapes.

Defect analysisⁱⁱ is pretty simple. First find the most common types of defects. I recommended doing Pareto charts based on the lifecycle phase the defect was introduced, the type of defect (algorithm, locking errors, non-initialized variables, etc.) and the module where the error occurs. Take the most common types and figure out the root cause of the defect, (ask WHY 5 times) and then make a recommendation on how to remove that root cause. We found one consistent problem: often the first recommendations are very generic, like “better reviews” or “better testing”. We required specifics: what should be done in the review, what types of tests need to be added, what training was needed.

After having each subsystem complete their recommendations, there were some common themes across the product that stood out.

First, the majority of defects were in the core operating system (what we call NCKL, Networking, Commands, Kernel, and Libraries). Historically, we had built our product inside out – putting together the core, testing it, and then layering other areas (user interfaces, system management software, and other middleware) on top of it. We would spend months getting the core to be stable enough to layer the other software.

50% of the defects were functional defects – things that should have been found by good functional or black box testing.

Another 10% were defects that only showed up under certain configurations. Engineers didn't have access to many of the configurations where problems occurred – at best these would be found in system testing, at worst, customers would find them.

Finally, we noted that developers only found about 30-40% of their own defects – most were being found by other development engineers or system test and 12% were found by customers. This was a very inefficient way to find defects.

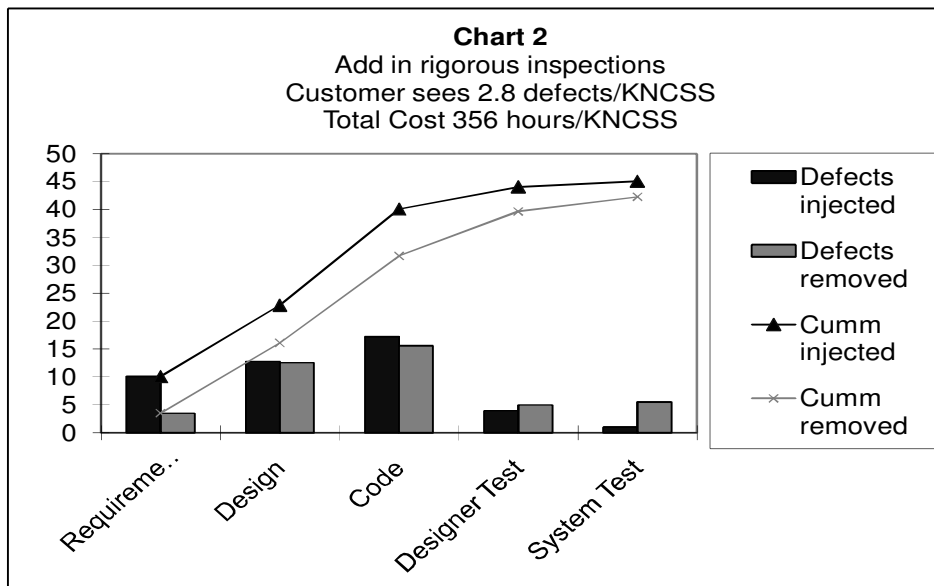
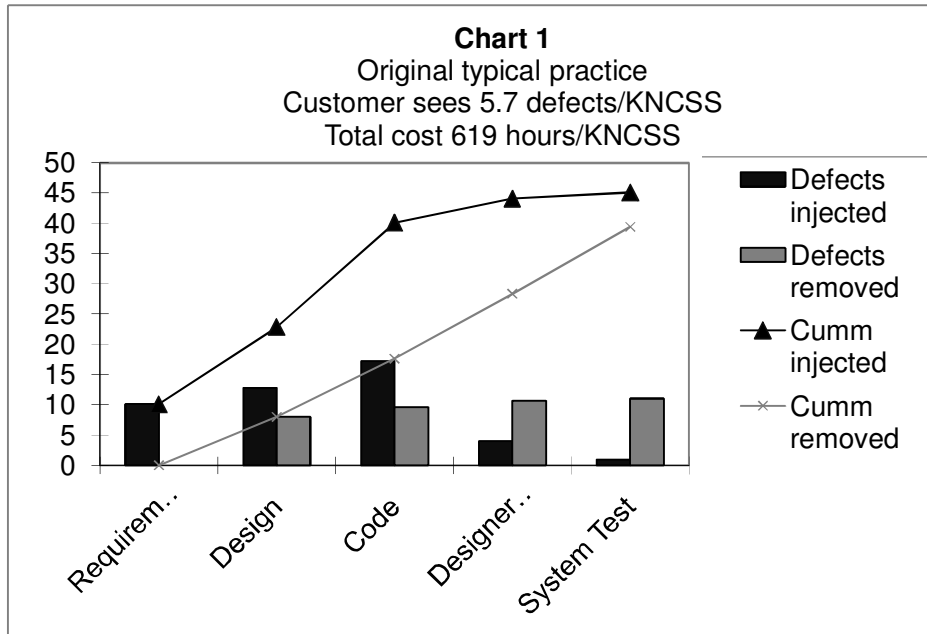
The Business Case for Management

Of course, if we were going to make major changes, we needed to show our management team that these changes would actually save them money. What follows are the actual justification slides that I used at the time. First, I used data from Capers Jonesⁱⁱⁱ to demonstrate that half of all defects are introduced before coding begins. Second, Capers also publishes the typical efficiency for various peer reviews, inspections, and test methods. One key point of his data shows that formal inspections are twice as efficient as informal review. For example, the median removal rate for a design review is that it will find 30% of the defects in the design, while an inspection will find 65% of the defects. New function testing will typically remove 30% of the existing defects. Of course, that new test will eventually become a regression test. Regression tests only find 15% of the remaining defects. This is to be expected – a new test tends to find new defects, rerunning that same test will only find whether a change elsewhere in the system broke something. Finally, we used a combination of our in-house data and Capers Jones^{iv} data to determine the typical hours it took to find and then fix a defect with each of these processes.

Using the data of the number of defects introduced and the time to find and fix a defect, and knowing our typical quality methods of the day: a design walk-through, a code check, writing new function tests, running existing regression tests, and having a system test, resulted in Chart 1. It shows the number of defects injected during each phase, and the number removed. The lines show the cumulative defects injected and removed. And, of course, the difference between the two lines shows the number of defects remaining in the code.

So given our current methodology, we were spending about 619 hours finding and fixing defects and shipping with about 6 defects per thousand lines of new code. Since the time to find and fix a customer found defect was well over 40 hours, not counting field time, shipping that many defects to customers was extremely costly.

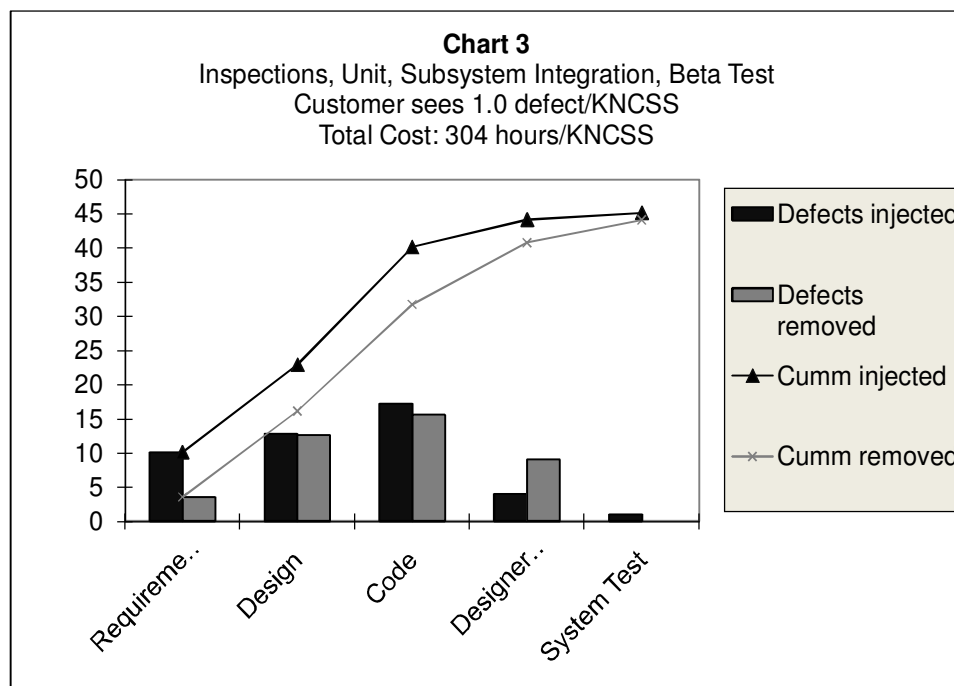
Chart 2 looks at inspecting everything (remember we typically had only design walkthroughs and code desk checks). What was the most convincing was that we could reduce the defects shipped customers by a factor of 2, while saving 42% of the labor. Most of the labor savings come from the fact it typically only takes ½ hour to fix a defect in an inspection versus multiple weeks if a defect is found by system test or by the customer.



These two charts convinced management that we should inspect all requirements, specifications, and designs; and retrain every engineer on inspections. The data wasn't as compelling for code inspections, so we left it up to the teams to decide whether to use code inspections or informal code reviews. As part of this effort, we began tracking metrics on peer reviews and created formal checklists for our inspections. Many of the labs also started using plan-do-check-act each quarter on their inspection and review processes.

Then we looked at beefing up our testing on top of the inspections. What if we did formal unit testing, system integration, and beta testing? As you can see from Chart 3, our labor only was reduced by 15% (it's more costly to develop tests than to do inspections) but our defect rate decreased by a factor of 3. This was sufficient evidence to our management that they were willing to invest in both equipment and test development to improve our testing. We now needed to develop a test strategy to determine exactly where we wanted to improve.

You'll remember our goals were to reduce the defects found after check-in by 2X every 18 months and to find more defects between check-in and release. With this data, we actually decided to change our internal goals to emphasize finding defects by the development team – because it would reduce the total cost. The goal was changed to have the development team find 90% of their own defects (through inspections and test), then system test (and other test groups) would find 90% of the remaining defects, leaving only 1% to be found by customers.



Creating a Test Model

So after building a strategy to improve our testing based on the fact that it would save us costs and significantly improve customer quality, it was then time to figure out our test strategy – exactly what were we going to improve. Our starting point was to document the current strategy. We did this by creating a process model of our testing. Then using our defect analysis, we built a target process model – what the testing strategy should be.

Current Model

Tests of subsystem		Tests of entire product		
Module Test	Subsystem Test	IC Test	System Test	Solution Test
No criteria	>95% pass on functional test on 1 SPU)	>95% pass on functional test on a couple configurations Run stress tests on small configuration	>99–99% pass on functional test (more varied and larger config) 96 continuous hours of operation on stress tests	0 defects running with key layered applications

Chart 4

Chart 4 represents the current model of our test strategy. Notice that it has two major subdivisions – testing we do on subsystems, and testing we do on the entire product. Development teams did module testing - the design engineer, knowing his design, did some testing to determine whether the module worked as he thought it should. Most engineers did this, but we had no clear expectations of what was required here. So in our future model we decided we should ask for unit testing, testing of less than 300 lines of code with branch flow goals.

We also had regression tests for every subsystem, which were primarily functional tests. The good news was that most of these tests were automated and run on a regular basis. We had also set criteria that they should always pass at least at a 95% rate. The bad news is that we really didn't know how much of the system these tests actually tested. (You'll remember that functional defects were the most common type of defect, so we knew that these had to be improved.)

We also had an Integration Cycle (IC) every 2 weeks where we put together the entire product. We would run most of the functional tests on these ICs and run some stress tests. The big difference between the IC test and the subsystem tests was that for subsystem testing we had the latest version of each subsystem and we tested on a larger set of configurations. For our stress tests, we measured what we called continuous hours of operation – how long the system would run before it hung or had a panic.

Finally, toward the end of a release, we would begin system testing. System testing would include many of the layered products as well as the UNIX operating system itself. We used the same functional regression tests, but in system test we had much larger variety of configurations, and in particular, this is where we would test on our large multimillion dollar computers. We tried to get to at least a 99% pass rate before we released the product. (Sometimes the tests would fail occasionally just because the test weren't written to handle all situations, so 100% pass wasn't practical.) Stress testing was also done on these larger configurations – with a goal to get two runs of 96 hours of continuous operation (CHO) in a row before we could release.

Finally, we had what we called solution test. Early on, this really consisted primarily of testing with HP layered applications and giving the product to our field engineers and Independent Software Vendors (ISVs) to test. You'll notice that our solution test didn't include any non-HP applications, nor did it reflect any typical customer configurations.

So we used our defect analysis to help us define our target model in Chart 5 (new tests are labeled in white). As part of our target model, we carefully defined the purpose of each kind of testing, its objectives and how we would measure success.

First, we looked at the big picture. Since over 60% of our defects were in the core product (NCKL), we actually added separate testing of this core to our test strategy. Since defects in the core affected everyone, we wanted especially to be sure that most of these defects were found before submittal. You'll see that we added three new test types to find defects quickly. Let's start at the top left and look at the reasons behind our chosen test strategy.

Module testing was formally renamed to unit test to indicate two things: a) we were expecting it to be done on smaller portions of code, not an entire subsystem. Typically, this would be 300 lines of code or less. b) We expected to achieve 100% functional and branch coverage by using a harness around this small set of code to be able to test all paths. Today, this is probably still our weakest area – largely because we didn't set up a mechanism to track that this was actually being done. It's also the case that we have so much pre-existing code (remember those 18 million lines of code), that it would be impractical to create unit tests on all that existing code.

Target Model

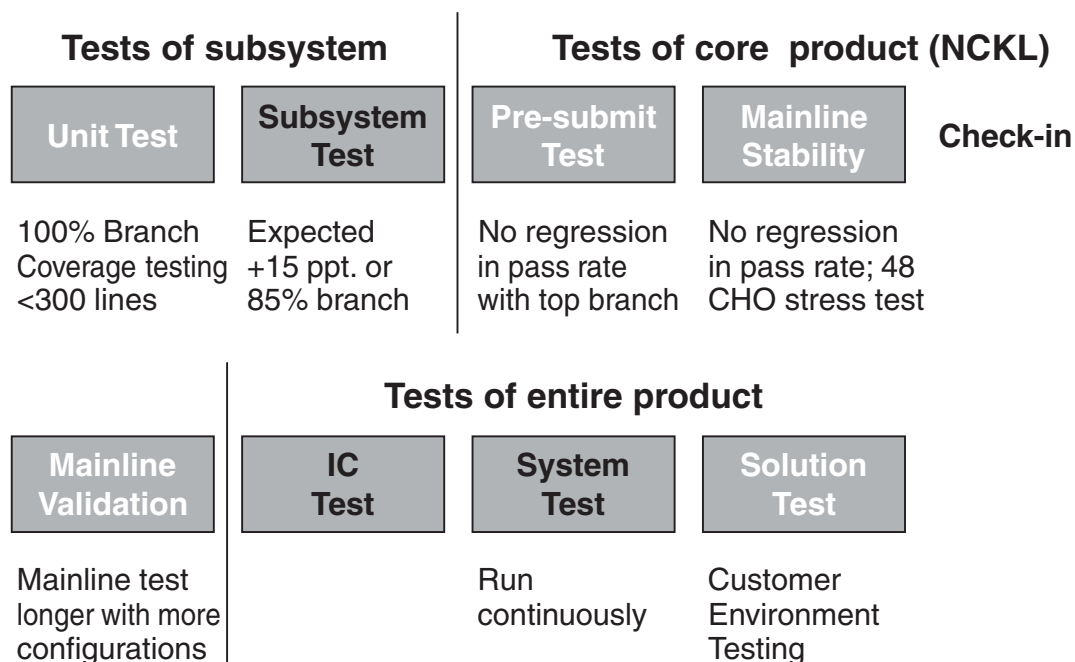


Chart 5

For subsystem testing, we added some criteria of goodness to our existing pass rate goal of 95%. We set goals to improve the functional and branch coverage of these functional tests. For new code, we expected 85% branch coverage and 100% function coverage. For existing code, we asked that for every major release, teams try to improve the coverage by 15 percentage points, from 50% to 65% coverage, for example.

Pre-submittal testing was a new test that was added and run by the developers. When they were ready to submit their code, they would run the functional tests against the top of branch for all the core modules. The goal was that there would be no regressions – all those tests that were passing before must continue to pass. This allowed us to slowly increase the pass rate of the entire system from 95% to 99.5%.

We then added a 48 hour test called mainline stability. Here we had a wider variety of configurations (including some of our high end SPUs) and ran the complete set of functional tests as well as 48 hours of stress test. If these tests failed, the check-in to core was rejected – we either backed out the submission or fixed it immediately.

After check-in we would continue to run our module testing in different configurations of networking and SPUs. Because differences in configuration accounted for 10% of our defects in our defect analysis, we wanted to increase the variety of configurations.

We continued to build the entire product every 2 weeks and run the same tests on it; this part of our test strategy remained relatively unchanged. Similarly our system tests didn't change much, except for one huge difference in the environment. Because our quality was higher, we actually could run these tests all the time, instead of waiting until the end of the release.

Our final change was to add solution test. We started running many more ISV applications on the system rather than just asking them to test. We also started building test environments that looked a lot more like our customers' environments. The test methodologies were enhanced to take on more of the customer style experiences: install/update, recover, data migration. The hardware configurations were expanded to fit our customer profiles, which typically consisted of 3 to 4 tier internet applications.

This has since been enhanced even further. We have representative customer application stacks for particular industries. We use trend-setting customers, those customers who tend to be the innovators in their industry and are stressing our systems the most. We've found that if we can get those applications stacks working properly, most customers in that industry will also benefit.

Results and Lessons for Others

In our first release with the new model, we reduced the customer-found defects by 6X, even though we added twice as many lines of code as the previous release. In that same release, we did achieve our goal that 90% of all defects were found by the developers. In our second release, we reached the goal of customers finding less than 1% of all defects. At the same time, we produced 30% more code/engineering month, reduced our backend testing time by 25% and reduced our current product engineering effort from 20% of our engineers to 13%.

The lessons for others are fairly clear:

1. Use formal inspections

If you aren't using formal inspections, this is the place to start. Inspections are well proven in the software literature as key to finding defects earlier, when they are less expensive to fix. However, in order to convince management of their value, you may have to generate your own return on investment data similar to what we did.

2. Use defect analysis to pinpoint improvements for your quality process.

Defect analysis builds its own support among the engineering teams because they decide how to fix the most frequent problems. Your defect analysis may also suggest changes in your test strategy. As you build the new strategy, each stage of testing should be well defined in terms of its purpose, objectives, and metrics.

The combination of these two techniques leads to both higher quality and productivity.

ⁱ “Management Commitment to Quality Requires Measures” by John Balza. PNSQC 2003

ⁱⁱ “Defect Analysis – a tool for improving software quality” by John Balza, PNSQC 2004

ⁱⁱⁱ “Software Quality, Analysis and Guidelines for Success” by Capers Jones, 1997. Pp 138-140 (10,000 FPs size projects)

^{iv} “Applied Software Management” by Capers Jones, 1991

I HAVE TWO MANAGERS?! : ONE COMPANY'S MODEL FOR A CONSULTATIVE TESTING TEAM AND MATRIX MANAGEMENT

Amy Yosowitz

Senior Information Technology Manager, Quality Assurance ■ Apollo Group, Inc.

(602) 387-7883 ■ amy.yosowitz@apollogrp.edu

ABSTRACT

Apollo Group, Inc., parent company of University of Phoenix, faces an interesting challenge in providing homegrown, feature-rich, user-friendly software to a staff of thousands who serve more than 350,000 students. With 30+ integrated applications being continuously developed by almost as many development teams within the Business Systems Division (BSD) of Apollo IT, we have chosen to go the route of having a consultative testing team. Our Software Quality Analysts (SQAs) have dual identities. Each SQA is a member of a larger QA organization where individuals learn skills, exchange ideas and business knowledge with other testers, and receive mentoring and direction from QA management. Each SQA is also a highly valued member of a development team where they interact with their development lead, developers, and business contacts. SQAs receive their direction from both their QA manager as well as their development team lead.

This paradigm has resulted in well-rounded SQAs who are fulfilled in their careers and have a deep sense of ownership over the applications they work with. With multiple management escalation paths, quality of our products does not get shortchanged. Our division has little to no “throw the code over the wall to QA” mentality, and development leads each have dedicated testers that they trust in and collaborate with closely in their day-to-day tasks. QA management spends much of its time mentoring, coordinating system-level testing efforts, and propagating best practices among SQAs and development teams.

This paper reports on the many aspects of this model, including: the roles and responsibilities of development leads, QA managers, and SQAs within our IT division; the elements involved, such as meetings, interactions, goal-setting, and reviews; the benefits and advantages of being organized in this manner; and the challenges we face.

BIOGRAPHY

Amy has over 10 years experience in the software field. She is currently a Senior Information Technology Manager for Quality Assurance at Apollo Group, Inc (the parent company of University of Phoenix), where she manages a group of 25+ software testers that focus on testing business applications. She was formerly a Senior Quality Assurance Engineer at Alogent Corporation (now Goldleaf Financial Solutions) and a Consultant at HBO & Company (now McKesson Corporation). During her tenure at Alogent, Amy was a lead tester on a state-of-the-art banking teller system. At HBO & Company, Amy performed implementation, testing, and design tasks for an innovative hospital information system. Amy holds a Bachelors of Arts degree in Mathematics with a French minor from Emory University in Atlanta, Georgia. She resides in Tempe, Arizona with her husband Rob and her toddler son, Blake.

BACKGROUND

Apollo Group, Inc. is the umbrella organization for a number of educational institutions, including its largest subsidiary, University of Phoenix. University of Phoenix is an institution of higher learning geared toward the working adult and offers associates, bachelors, masters, and doctoral programs. University of Phoenix has ground campuses in almost every state in the United States as well as campuses abroad. One half of University of Phoenix students attend the Online campus, where coursework is done in an asynchronous fashion through online forums. Other Apollo institutions include the Institute for Professional Development, Center for Financial Planning, Western International University, and Meritus University. Apollo has over 350,000 active students, more than 400,000 alumni, approximately 25,000 faculty members, and approximately 20,000 staff members.

Apollo Information Technology (IT) is the department of Apollo that develops and tests both internal and external-facing software applications for Apollo's subsidiaries, as well as provides technical support, hardware support, database administration, network infrastructure, and other services to respond to all other technical needs of the company. The software area is split into three divisions: Business Systems Division (BSD) for most internal-facing software that the business employees use, Product Development Division for external-facing software that students and faculty use, and Finance/Human Resources (FHR) Division for finance and human resource applications.

This paper will focus on the software testing model used by the Business Systems Division (BSD) of Apollo IT. BSD has responsibility for over 40 different home-grown business applications. These are in varying stages of maturity; some are legacy applications, some are new applications developed within the past five years, some are brand new applications. All applications continue to receive enhancements and bug fixes on a regular basis to respond to the changing needs of our business. The development staff is split into small teams of typically ten or less developers, with each team taking responsibility for one or more closely-related applications. There is one development team lead ("dev lead") per team.

TESTING ORGANIZATION

The testing team, often referred to as the Software Quality Analyst (SQA) team, works off of a consultative model and has a matrix management reporting structure. Our team currently has more than 25 SQAs who, from a human resources perspective, report up through to me, a senior QA manager. I have another QA manager reporting to me who helps to share the load of this large group.

It is my responsibility, along with my fellow QA manager, to assign permanent testing resources to each development team. It is in this way that we have a consultative model; much like a consulting firm provides the right resources for a job, QA management seeks out the right resources (as well as the right number of resources) to fulfill the specialized testing needs of each development team. It is important to note the differences between our model and another popular model for testing teams:

<u>Common Model</u>	<u>Apollo Model</u>
Pool of resources with generally similar product and testing knowledge	Pool of resources, each with varying testing experience and each with specialized product/business knowledge
Resources are assigned relatively short-term tasks, then when complete, are assigned to new testing tasks	Resources are each dedicated to a development team and perform all testing that team needs on an ongoing basis
QA management distributes tasks among SQAs	QA management mainly serves as a mentor and advocate for quality. SQAs, along with their respective dev leads, delineate the testing tasks that need to be done for a particular release cycle of each product.
SQAs all sit together in a central area	Each SQA sits embedded within the development team they serve
Some SQAs create tests, others execute those tests	Each SQA is responsible for both test case creation and execution for their product

When assigning an SQA permanently to a development team, several items are taken into consideration, including: communication style, business knowledge, technical skill, and the SQA's ability to work independently. Different development teams have different work styles and expectations of their testers, so QA management tries to make the best fit possible.

The other key element to our model is matrix management. The SQAs do all report up through me in the HR reporting structure, but through a "dotted line" in the reporting structure, they also report to the dev lead for their development team. The dev leads, who have overall responsibility for their respective application(s), are the ones that the SQAs interact with on a day-to-day basis. The SQA works with the dev lead (as well as the developers) on understanding requirements, creating estimates, defining the testing approach, test cases, troubleshooting, and raising system issues. The QA manager has a different relationship with the SQAs. The QA manager does more general oversight, ensuring that the SQAs are doing the right things to test their applications. SQAs are mentored by QA managers in areas of testing (including design and execution), technical skills, communication skills, system integration points, and time management. From the point of view of the SQA, the matrix management model provides the best of both worlds: a dev lead who knows the ins and outs of their particular product, and a QA manager with a higher level view of all of our systems.

Matrix management also has the benefit of providing multiple escalation paths for issues, which is always a difficult area in organizations where testers report directly to development managers. If a dev lead is having any sort of problem with their SQA, they can escalate to the QA manager, who has the expertise to mentor and work with the SQA. Development management typically does not have the mindset or the skill set to be able to help an SQA do their job effectively and efficiently. In addition, if an SQA feels the dev lead is repudiating the SQA's ideas and attempts to ensure a quality product (for example, "I know you haven't had enough time to test, but we need to send this to production anyway."), then the SQA can escalate to the QA manager, who can examine the situation and work with the dev lead to assess risk and arrive at the best possible outcome.

ROLES

SQAs, QA managers, and development leads all have important roles in making this model work. Here are some further details about these roles and their responsibilities:

SQA (Software Quality Analyst)

- Often functions as a "jack of all trades" – part tester, part business analyst, part developer. A well-rounded SQA has excellent analytical and logical thinking skill and therefore first-rate test case

creation skill. The SQA also understands Apollo's business model and is skilled at understanding and dissecting requirements, finding both intended and unintended consequences of new features. Lastly, the SQA has good technical skills with the ability to automate tests as well as write database queries to validate data and create and search for needed test data.

- Helps the dev team to understand the user perspective as well as the core business reasons behind software changes
- Decides, with support from the dev team, what testing is necessary for new changes and bug fixes. Compiles all test cases and test data.
- Maintains and executes manual and automated regression test scripts
- Creates and maintains automated smoke test scripts, used both in the QA environment as well as for production release validation
- Logs bug reports and helps the dev lead to prioritize any issues found and determine any "must fix" items for a particular release
- Makes recommendations to the dev lead as to whether a release is ready to go to production (or not), but does not have the final decision
- Generally has responsibility to protect the production environment
- Often has responsibility to organize, script, and/or perform product demonstrations for business users
- Calculates Defect Detection Rate (DDR) metrics on a regular basis
- Directly observes end users at least a few hours per month

QA Manager (Quality Assurance Manager)

- Mentors SQAs in areas of testing (including design and execution), technical skills, communication skills, system integration points, and time management
- Helps SQAs reach beyond the everyday to provide the most value to the organization
- Mediates conflicts between SQAs and development
- Serves as the advocate for the SQA and for quality software in general
- Performs all administrative tasks for SQAs, including: performance reviews, access approvals, hiring, and informing regarding company news
- Addresses any testing or SQA concerns the dev lead raises
- Organizes training and mentoring opportunities
- Manages the SQA team to work toward larger initiatives like quarterly goals
- Coordinates testing large IT projects, such as database moves, upgrades, and implementations for new user bases
- Shares knowledge between development teams. Each dev lead is given quite a bit of freedom to organize and run their team as they think is best. Because of QA management's exposure to many team's practices and methods of solving problems, they are in a valuable position to spread best practices, process improvements, and problem solutions among various development teams.
- Understands the high-level system architecture, data sharing between applications, and dependencies between products. Upon hearing of new features or new types of data interactions in one application, interprets these items and makes recommendations as to what other applications may need testing in conjunction.

Dev Lead (Development Team Lead)

- Serves as the leader and single point of contact for an application or a small group of related applications
- Has ultimate accountability for the quality and success of their application(s)
- Represents the application to the corresponding business unit(s)

- Manages and organizes developers and their work
- Ensures adherence to processes and standards that the team uses, including coding standards, software development lifecycle activities, and other best practices
- Interacts with the SQA on day-to-day activities
- Prioritizes, along with the SQA, the testing of new features, bug fixes, and regression tests, as well as prioritization within each of those areas, to be in line with risks, business priorities, and various uses of the application.
- Regularly reviews regression test cases for coverage and depth
- Makes the final go/no go decision on production releases, using information provided by the SQA

CORE ELEMENTS

There are some principal practices that Business Systems Division of Apollo IT follows in order for this model to be successful.

- The QA manager and each SQA have a monthly one-on-one meeting for one hour.
 - This discussion can include: updates on how their dev team is functioning, follow up on goals and testing initiatives, any obstacles the tester is facing, professional development plans, review of test cases and test executions, metrics, upcoming releases and functionality being added, participation in system test efforts. It is integral to have this one-on-one time; with the sheer number of SQAs sometimes it is the only time these individuals can talk one-on-one.
- The QA manager and each dev lead have a monthly one-on-one meeting for 30 minutes.
 - This discussion can include: testing, process, and general SDLC (Software Development Life Cycle) challenges, upcoming functionality being implemented, a review of the SQA's job performance – where they are succeeding and areas that need improvement, suggestions as to how the QA manager and the SQA can better help that team to succeed, upcoming multi-application initiatives.

→Both of these one-on-ones help the QA manager to have a high-level view of all systems and to be aware of changes in individual applications. The QA manager can then identify both software and data dependencies and pull in multiple testers to collaborate in testing when needed.

- The SQAs and their respective dev leads are encouraged, but not required, to have a one-on-one monthly.
 - This closes the SQA→dev lead→QA manager communication loop and allows the dev lead and SQA to discuss things that may fall outside of normal day-to-day discussions, such as overall test strategy and automation progress.
- Each development team, including developers, SQAs, and the dev lead, has a daily stand-up meeting for 15 minutes per day.
 - This goes along with agile best practices, and allows the team to track progress and issues from day to day. We have found it quite essential for good team communication. Teams that forgo this practice, even for a short while, note that they feel out of touch with each other.
- All BSD SQAs and QA managers have a mandatory weekly team meeting for one and one-half hours per week (or less time, as needed).
 - These meetings typically start out with announcements, which can include:
 - personnel changes (SQAs coming, going, or switching development teams)
 - company news
 - IT department news
 - a “thank you” session where managers thank SQAs who have helped out in a special or extraordinary way, as well as SQAs thank each other publicly for assistance, mentoring, or a good job on collaborative testing

- a “bug of the week” session where SQAs bring up a unique bug that was found that week. It could be a serious bug that impacted our business, a trivial bug that was funny, a bug that was found by accident, a bug found using an interesting testing methodology – anything of note to other testers. We discuss how the bug was found, the impact, and how the dev team dealt with it. We encourage an open forum where SQAs admit their mistakes and share learning experiences with other SQAs.
- There is also a general topic for each meeting. These can include:
 - An SQA presenting on the application that they test. This allows each team member to obtain knowledge on applications other than their own. Since our applications are all intertwined, this is important for testing purposes. It also helps everyone understand our business better.
 - A technical presentation on HP Quick Test Professional (QTP) or HP LoadRunner (LR)
 - A team building exercise or fun activity
 - Detailed review of a best practice or a specific SQA responsibility, such as metrics gathering or bug reporting
 - An SQA presenting on something learned from a recent conference
 - A focus on another area of IT, such as business intelligence, data warehousing, external-facing applications, support, or database testing
- All SQAs in all divisions and all SQEs (Software Quality Engineers, more technical testers) have an All SQA/SQE meeting for one hour each month, led by the director over QA. Topics are similar to those of the SQA weekly meetings.
- Most development teams are on two-week release cycles. This means that a new software version, with new functionality and bug fixes, is placed into production every two weeks. Development teams do this in one of two ways, depending on the nature of the application:
 - Overlapping cycles – A release goes to production. The next day, the SQA receives a build of the next version of the application to begin testing. While the SQA is testing this version for two weeks, some developers are dedicated to fixing bugs in that version, and some are already coding for the next version in a different branch. The version that the SQA is testing goes to production after two weeks of testing, then the cycle repeats.
 - One week/one week cycles – A release goes to production. The developers spend the next week coding for the next version of the application. While this coding is going on for a week, the SQA is catching up on automated scripting, understanding upcoming business requirements, adding to the regression tests, and doing spot tests on developers’ builds. One week after the previous release, the SQA receives a build of this new version and has one week to test it. Typically, most developers are focused on bug fixing during this week. The version that the SQA is testing goes to production after one week of testing, then the cycle repeats.
- The QA manager performs reviews on each SQA in the form of semi-annual and annual reviews. For both types of reviews, the QA manager obtains extensive feedback from the dev lead that the SQA works with to create the review. For the annual review, feedback from peers (fellow developers, SQAs, management, and sometimes business representatives that this SQA works closely with) is also solicited in the form of an anonymous peer feedback sheet. At each review point, general and professional development goals are set for the SQA including both project-based and personal career goals. Goal setting is a collaborative exercise between the SQA, QA manager, and dev manager.
- Each SQA gathers two flavors of metrics for each release after it has been through QA and also completed its production run (i.e. it has been replaced with another version in production):

- Defect Detection Rate metrics (DDRs) measure the number of defects found in the QA portion of the lifecycle (pre-production) compared to the total number of defects found in that release (including defects found post-production). These are also broken down by severity to better interpret the numbers. This gives the SQA, dev lead, and QA manager a way to measure how well the SQA did at protecting the production environment from significant defects.
- Day-by-day metrics measure the number of defects found each day a version is in QA and in production. It is a goal of all SQAs to provide information about the quality of a build as early on in the cycle as possible, so that the development team has adequate time to react and fix things. This measures how quickly SQAs were able to provide feedback to their dev teams.

→Both types of metrics often trigger interesting discussions of both QA and development practices within a team, whether it is the need for more unit tests, better prioritization of testing, more regression test cases, or development strategies for lowering the total number of bugs found each release.

- The SQAs sit among their development team. This greatly aids in communication, and the developers think of the SQA as a member of their team and not an unknown entity that they throw code over the wall to.
- Most SQAs attend all business meetings that the dev lead has with their corresponding business unit. This allows the SQA to see the decision-making process and ensure the final software changes do indeed fit the users' needs and intents. SQAs also often serve an important business analyst type role in these meetings, translating items between the developers and the business users.
- SQAs do not make the final decision as to whether a build goes to production; this is a decision left to the dev lead, as they are ultimately responsible for the success or failure of the application and its quality. SQAs do have the important responsibility of making a recommendation as to whether the build should go to production or not, and have to gather supporting data to back up this recommendation.
- Business Systems Division (BSD) has a general practice of moving employees between products every few years. This is especially true in the case of the QA department. QA management feels that after two or more years testing the same application, even the best tester can have blinders on to important or interesting bugs. Also, the tester is no longer challenged as much as they should be. In addition, having an SQA being a product expert for too long can result in too much of a safety net for the development team; developers may start taking shortcuts or using poor development practices because they figure the experienced, expert SQA will catch anything they mess up. Sometimes SQA swaps are initiated by the SQAs themselves who want to test something different and/or learn a new part of the business; sometimes they are orchestrated by QA management among two or more people we think need a new challenge. Transitions between testers usually occur over the course of a few months. While sometimes transitions are tricky, in the long run, they typically benefit both development teams. Each dev team gets an SQA with a fresh set of eyes on their application, as well as someone who understands other parts of our business. SQAs also know what development cycle practices have and have not worked on their previous team, and they can bring this experience and some new ideas to their new team.
- The BSD directors define goals each quarter for both development and QA that are good practices that can get left behind in the day-to-day work of building software. Recent quarterly goals for QA have included: ensuring all test executions are done within Quality Center, building system level tests for business processes that involve multiple applications, and improving production application monitoring.

- For most of our applications, we try to stick to a ratio of one SQA for every three application developers on a team.

MISCELLANEOUS ELEMENTS

Here are some other elements of our environment that are not all directly related to testing, but important to note:

- We have a team of application Software Quality Engineers (SQEs). These are more technical testers that perform static code analysis, load testing, continuous integration tasks, and other white box testing techniques. They use a similar consultative testing and matrix management model to that of the SQA team.
- We have a team of database SQEs. These are testers that perform testing on database code including stored procedures, triggers, packages, and data transfers. They have a slightly different model of serving their customers, but it still can be considered matrix management and consultative testing.
- We have a team of database developers that are separate from application developers. Many applications share databases with other applications. These developers are somewhat independent from the development teams, but obviously have a need to frequently interact and collaborate with application developers.
- We have a team of Human-Computer Interaction (HCI) specialists. These individuals help produce easy-to-use and intuitive user interfaces for all applications. They are also set up in a somewhat consultative manner, although they do not generally follow the matrix management model.
- With a few exceptions, we do not have traditional business analysts for each business unit or application. In many ways, the dev lead and the SQA perform the business analysis tasks for a team. This has benefits and disadvantages. It puts the development team much closer to its business and causes our applications to align quite well with business needs, but it also puts an extra workload on dev leads and SQAs who have other primary focuses.
- We follow a mix of various agile methodologies.

BENEFITS AND ADVANTAGES

BSD enjoys many benefits of this consultative testing team and matrix management model:

- Each SQA has a great sense of ownership, loyalty, and belonging to their development team and their product. This results in devoted SQAs who always go the extra mile to ensure the quality of the software and make sure it fits the users' needs.
- SQAs are the subject matter experts and are very much go-to people on their respective development teams. They greatly enjoy being able to give this amount of value and get a lot of fulfillment out of their job.
- SQAs are typically the "right hand man" for their development lead. The dev leads have a high sense of trust in their SQAs, and often depend on them to help lead the team when the dev lead is out of the office. This is wonderful when compared to IT organizations where development sees QA as a roadblock to their success. In Apollo IT, it is just the opposite – QA is an essential depended-upon part of the organization.
- The SQAs get a great variety in their jobs: the logic/analysis/test case creation aspect, the technology/automation/database aspect, and the interaction with business and requirements. This makes their job a lot less boring than if they just wrote and/or executed test cases all day.

- Quality does not get shortchanged due to the multiple escalation paths SQAs have (QA and dev management) for getting issues addressed. This is an advantage over a setup where QA reports directly to development.
- With SQAs assigned to work with a team permanently, the team develops relationships and gets the chance to really “gel.” There is little to no “throw the code over the wall” mentality. New employees often find this remarkable.
- Having typically one to three SQAs work for a single small development team helps with determining accountability for testing errors on a particular product.
- We have very few examples of contentious relationships between QA and development.
- As a consultative testing team, we feel having dedicated SQAs provide their services to development teams provides better customer service and visibility to our development leads. This is an advantage over having a centralized QA team where tasks are distributed.

CHALLENGES

Any organization that uses this consultative testing and matrix management approach will face some difficult aspects to manage.

- While we generally like to allow development teams and SQAs to determine their own best and most efficient ways of doing things, some items must be standardized. This can include standardization of test case documentation, automation scripting standards, and even a standardized telecommuting policy. With each SQA as part of a different development team with different practices and conventions, it makes it difficult to standardize on most things. QA management also finds it difficult to enforce standardization with such a diverse team.
- QA management also finds it difficult in our model to monitor the productivity of the SQAs. Some of this is related to difficulties with standardization. With so many applications and typically only one or two SQAs per application, it is hard to determine if the SQA is slow or fast, efficient or inefficient, productive or unproductive with the time that they have to test.
- In our environment, it is just about impossible for one person to have in-depth product knowledge and business knowledge in all areas. In many cases, SQAs are experts in their own area but do not know about other areas, especially the upstream or downstream applications related to their own application. We encourage informal knowledge sharing as well as formal knowledge sharing through application presentations and organized system tests. In the end though, getting each SQA to know their application as well as other applications in their same business area and/or ones that theirs shares data with is often challenging.
- With typically one SQA per application, we face many challenges when SQAs take vacations or are out of the office unexpectedly. There is no natural person who can perform their responsibilities to their level of expertise in situations like this. Management has set up a system of application backups, where each SQA is a backup for one or more applications. When the main SQA for an application is out of the office, their backup is there to help their development team with testing efforts. This is by no means an ideal solution. When someone is out of the office, their backup must split their time between their main application and their backup application. This requires both development teams to slow down their development somewhat since neither has a full-time SQA during the period. Also, the development team has a difficult time putting their trust in a backup SQA that they do not know and that does not know their application nearly as well as their regular SQA. In addition, it is a challenge to make sure backups stay current on the features of the applications that they back up.
- With SQAs each specializing in one product or a group of small products, they can tend to have blinders on and focus their testing only on their application. Apollo often has business functions that

span multiple applications, and up until recently, these rarely got tested as a system. We have started putting together system tests for these important functions. All SQAs that work with applications involved with this business function meet over a period of months to design both a manual and automated test that covers the main scenarios and business workflows. This has been a slow but successful effort so far.

- We have recently faced an issue where, despite our best time management efforts, the SQAs are not able to keep the automated regression test suite up to date. We have fallen behind in continuously creating automated regression tests for new features, and many existing automated tests have been broken for quite a while. With the constant push to get new releases out every two weeks, automation ends up being deprioritized. It is important, but not urgent. We are considering changing our model so that we have a separate small team of automation specialists that write and maintain automated regression tests for all of our applications. We are also looking toward keyword-driven automation as a methodology that could work in this new model.

CONCLUSION

Using this consultative testing and matrix management model has been very successful for Apollo IT's Business Systems Division. Our SQAs feel very valued and fulfilled in their jobs, and as a result we have very little turnover. Development team leads love the fact that they have a dedicated tester that they can always depend on. We have an organization that from the top down values quality and intuitively easy-to-use software applications that scale with our growing business. The Software Quality Analysts at Apollo Group are excited to take part in creating innovative software that fulfills our business goal of serving students well.

The Myths of Rigor

James Bach

I keep hearing that more rigor is good and less rigor is bad. Some managers who've never studied testing, never done testing, probably have never even *seen* testing up close, nevertheless insist that it be rigorously planned in advance and fully documented. This is a cancer of ignorance that hobbles our craft.

I have done some of the most rigorous testing any tester will do in a career, as part of winning court cases. Let me tell you, extreme rigor costs a lot of money and brings a laser-like focus to testing. But if you were in a dark room, you wouldn't use a laser scanner to find your car keys, would you? A soft light would do.

In this talk I will describe and dispel various myths of rigor, so we can apply rigor without obsession or compulsion, and let our testing be flexible and inexpensive, too.

***James Bach** has been a test manager or consulting tester since Apple lured him from a programming career in 1987. He spent about 10 years in Silicon Valley before going independent and traveling the world teaching rapid software testing skills. James passion is to teach testers to think, which is why he strongly opposes thoughtless programs such as ISTQB tester certification (and every other certification program currently out there). He is the author of **Lessons Learned in Software Testing**, and a new book: **Secrets of a Buccaneer-Scholar**, which describes his approach to self-education.*



The Myths of Rigor

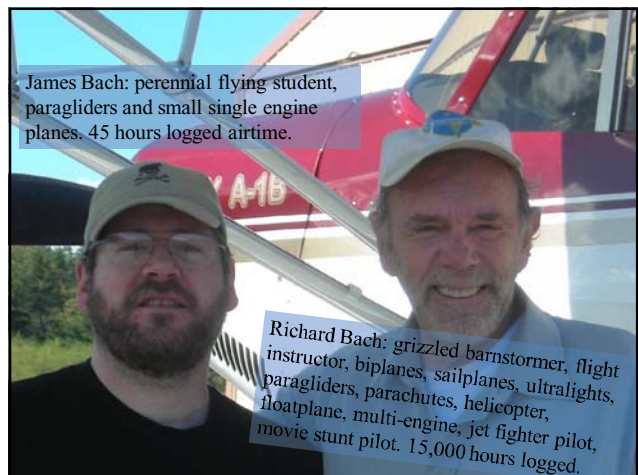
James Bach, Satisfice, Inc.
james@satisfice.com
www.satisfice.com

Rigor

- Oxford English Dictionary
 - The strict terms, application, or enforcement of some law, rule, etc.
 - Strictness of discipline, etc.; austerity of life; an instance of this.
 - Strict accuracy, severe exactitude.

Key Idea:

There can be no rigor without a standard of correctness.



James Bach: perennial flying student, paragliders and small single engine planes. 45 hours logged airtime.

Richard Bach: grizzled barnstormer, flight instructor, biplanes, sailplanes, ultralights, paragliders, parachutes, helicopter, floatplane, multi-engine, jet fighter pilot, movie stunt pilot. 15,000 hours logged.

Flying the Husky Rigorously

- After flying with me a few days, Dad says “*What are the checklists for flying the Husky?*”
- I researched and created the checklists. Dad reviewed them and offered corrections.

START	
Prestart	
Preflight	COMPLETE
Hydraulic System	DEPRESSURIZE
Seat Belt/Harness	FASTENED
YFD	ON
Water Rudder	UP
Gear Selector	AS REQUIRED
Auxiliary Switch	OFF
Alternator	ON
Strobe Switch	ON
Mixture	RICH
Propeller	FULL INCREASE
Carb. Heat	COLD
Mastor Switch	ON
Throttle	CRACKED
Primer	AS REQUIRED
Engine Start	
Brakes	ON
Propeller	CLEAR
Magneto Switch	START
Throttle	AS REQUIRED, THEN IDLE
Alternator	ON
Auxiliary Switch	ON
Oil Pressure	CHECK

Dad's Corrections

- Some changes I understood.
- Some changes I didn't understand.
- Some changes I thought I understood but didn't.
- One change he made was wrong. (I caught it!)

Key Idea:

A standard isn't crystallized skill or action. It must be interpreted and may be incorrectly interpreted.

Flight Testing

- Using the landing checklist forced my thoughts into a fixed linear sequence.
- It took me out of my rhythm. *Timing and coordinated action are important in landings.*
- I tried to use the checklist in a non-sequential way, but that led me to forget things entirely.

WATER LANDING	
Water Landing	
Gear Call	"This is a WATER landing. Gear up for a water landing."
Gear Selector	UP
Gear Check	"The left wheels are UP, the right wheels are UP, the wheels indicate UP for a water landing."
Water Rudder	UP
Carb. Heat	HOT
Mixture	RICH
Throttle	REDUCE
Air Speed	70 MPH
Flaps	AS REQUIRED
Propeller	FULL INCREASE
Landing Gear	RECHECK UP
Post Landing	
Flaps	UP
Carb. Heat	COLD
Water Rudder	AS REQUIRED

"Dad, we should share these checklists on the Husky forum!"



"I suppose you could, James, but *Husky pilots don't use checklists.*"

Value of the Lists

- Excellent for quick review prior to the flight.
- The engine start checklist works great as is.
- Good tool for discussing and analyzing flight procedures.
- Creating them helped me learn.
- Using them without “following” them helps me fly better... *The next step is to memorize them for random access, experiment with different procedures, and practice, practice, practice.*

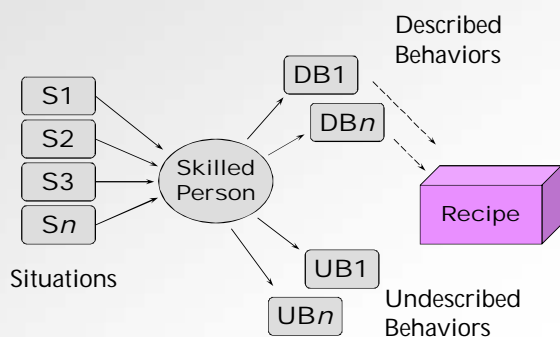
Good Rigor

- Doing good things at a good time in a good way without too much trouble.
- It must not cost too much to adopt, employ, or maintain in the face of change.

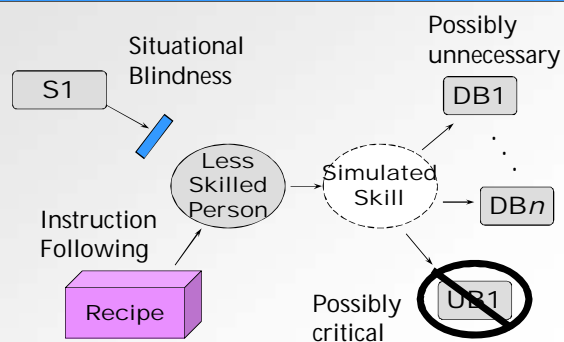
Bad Rigor

means corporate obsessive-compulsive disorder: following instructions without understanding them or being properly supervised, causing wrong or wasteful action.

The Skill Simulation Problem



The Skill Simulation Problem

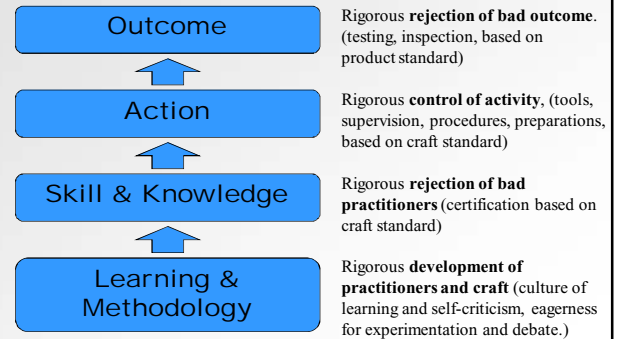


Coaching Rigor

- James: “Woo hoo!”
- Richard: “Not enough left rudder. Steep turns at low altitude must be coordinated or we risk an unrecoverable asymmetric stall, especially since we’re flying heavy.”
- James: “I *felt* like I had enough rudder.”
- Richard: “You didn’t. Practice feeling it. Let’s head over to Suchia and you can do some Dutch rolls.”



Where to Apply Rigor?



Unless you know what is the right thing to do...

Doing it rigorously will be reckless and irresponsible.

Doing it rigorously will retard your learning.

Premature standardization has harmed our craft.

The Myths of Rigor

- Adding rigor makes a process better.
- Engineering processes should be rigorous.
- Rigor is all or nothing.
- Documentation, metrics, scripted procedure, etc. is rigorous.
- Exploratory testing, agile, etc. is NOT rigorous.
- Playfulness and loosely defined methodology is not a route to rigorous processes.
- The route to rigor is for people to stop resisting and follow instructions.

Myths Adjusted

- Adding rigor make a process less flexible.
- Engineering processes should be reasonable in context.
- Rigor is all or nothing or anything in between.
- Documentation, metrics, scripted procedure, etc. might be bad rigor.
- Exploratory testing, agile, etc. might be good rigor.
- Playfulness and loosely defined methodology may be essential to developing excellent rigor.
- A necessary condition for good rigor is for people to be passionate about learning their craft.

In software development
various communities have
competing views of rigor.

*I rigorously pursue
what I believe is excellence
in software testing.*

Therefore,
I CANNOT accept “best practices,”
I CANNOT accept bad certification,
I MUST oppose bad standards,
I hope you will do these things, too.

START

Preflight	COMPLETE
Hydraulic System	DEPRESSURIZE
Seat Belt/Harness	FASTENED
PFD	ON
Water Rudder	UP
Gear Selector	AS REQUIRED
Avionics Switch	OFF
Alternator	OFF
Strobe Switch	ON
Mixture	RICH
Propeller	FULL INCREASE
Carb. Heat	COLD
Master Switch	ON
Throttle	CRACKED
Primer	AS REQUIRED

Brakes	ON
Propeller	CLEAR
Magneto Switch	START
Throttle	AS REQUIRED, THEN IDLE
Alternator	ON
Avionics Switch	ON
Oil Pressure	CHECK

Husky 42HY

LAND TAKEOFF

Brakes	CHECK
Controls	FREE & PROPER MOVEMENT
Instruments	CHECK
Fuel Selector	ON
Trim	SET
Runup	THROTTLE 1800 RPM
	PROPELLER CYCLE
	MAGS RIGHT, LEFT, BOTH
	CARB HEAT CHECK
	THROTTLE IDLE
Flaps	SET
Wind	CHECK
Sky	CHECK TRAFFIC

Flaps	UP
Gear Selector	UP
V_x	65 MPH
V_y	73 MPH
V_{fe}	80 MPH

Husky 42HY

LAND LANDING

Gear Call	"This is a LAND landing. Gear down for a land landing."
Gear Selector	DOWN
Gear Check	"The left wheels are DOWN, the right wheels are DOWN, the wheels indicate DOWN for a LAND landing."
Water Rudder	UP
Carb. Heat	HOT
Mixture	RICH
Throttle	REDUCE
Air Speed	70 MPH
Flaps	AS REQUIRED
Propeller	FULL INCREASE

Flaps UP
Carb. Heat COLD

Husky 42HY

WATER TAKEOFF

Controls		FREE & PROPER MOVEMENT
Instruments	CHECK	
Carb. Heat	COLD	
Fuel Selector	ON	
Instruments	CHECK	
Trim	SET	
Surface Traffic	CHECK	
Air Traffic	CHECK	
Wind & Wakes	CHECK	
Water Rudder	UP	
Flaps	SET	
Propeller	FULL INCREASE	

Flaps UP

Vx 65 MPH
Vy 73 MPH
Vfe 80 MPH

Husky 42HY

WATER LANDING

Gear Call	"This is a WATER landing. Gear up for a water landing."
Gear Selector	UP
Gear Check	"The left wheels are UP, the right wheels are UP, the wheels indicate UP for a water landing."
Water Rudder	UP
Carb. Heat	HOT
Mixture	RICH
Throttle	REDUCE
Air Speed	70 MPH
Flaps	AS REQUIRED
Propeller	FULL INCREASE
Landing Gear	RECHECK UP
Flaps	UP
Carb. Heat	COLD
Water Rudder	AS REQUIRED

Husky 42HY

EMERGENCY

Gear Handle	DOWN
Gear-Motor Circuit Breaker	PULL OUT
Hand Pump Gear Selector	SELECT (forward for UP, aft for DOWN)
Hand pump	PUMP TO LOCK (~100)
Hand Pump Gear Selector	NEUTRAL
Gear-Motor Circuit Breaker	PUSH IN

Husky 42HY

SCORE ONE FOR QUALITY!

USING GAMES TO IMPROVE PRODUCT QUALITY

Joshua Williams

joshw@microsoft.com
(425) 703-1059
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Ross Smith

rosss@microsoft.com
(425) 706-3982
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Dan Bean

danbean@microsoft.com
(425) 703-4238
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Robin Moeur

robin.moeur@gmail.com
(425) 538-7669
2552 E Roanoke Street
Seattle, WA 98112

ABSTRACT

In this paper we describe how using a game can improve both the quality of a product and employees' quality of life as well. We call this kind of game a "Productivity Game".

Productivity Games, as a sub-category of Serious Gamesⁱ, attract players to perform work that humans are good at, but currently computers are not. Although computers offer tremendous opportunities for automation and calculation, some tasks, such as analyzing images, have proven to be difficult and error-prone and therefore lower the quality and usefulness of the output. For tasks such as this, human computation can be much more effective. Additionally, by framing the task in the form of a game, we are able to quickly and effectively communicate the objective, and achieve higher engagement from a community of employees as players of the game.

We showcase a real Productivity Game taken directly from the Windows development process to highlight this engagement and its benefits. The "Windows Language Quality Game" encourages native language speakers to perform the job of traditional software localizers and enhances an otherwise difficult and expensive business process with a "serious game". This has resulted in players who enjoy the opportunity to participate and contribute. It has also resulted in a cost-effective way to improve the quality of native-language editions of Microsoft Windows.

The use of Productivity Games has broad implications across how employees are managed, and how employers communicate organizational objectives to their staff. Games in the workplace can be used to augment leadership, and can be more relevant, applicable and engaging to employees.

1 INTRODUCTION

The global business challenges of the 21st century require creative approaches and innovative solutions. Traditional methodologies for solving problems are evolving to create hybrid solutions that embrace new collaborative roles for humans and their use of computers. Technology is facilitating these hybrid solutions by enabling a large number of humans to focus on a problem and then easily aggregate their input. This has opened up the opportunity to innovate and creatively solve many business challenges.

In tandem, a generation gap has begun to appear between the established workforce and the Gen-Yⁱⁱ and Millennial generations which are now filling the ranks of young employees and college hires. This younger generation brings its own priorities, communication patterns and perspectives to the workplace, as have previous generations, but in this case the gap is larger, and the challenges even greaterⁱⁱⁱ. This younger generation is often referred to as “the gamer generation”, as video games have been central to their lives^{iv}. The influence of games on their expectations of work and life cannot be underestimated. So, leveraging games to engage this generation seems an obvious path to increasing engagement of young employees. This is not to say that games only apply to the younger generations. Employees of all ages find games engaging and fun.

The challenge comes when creating a hybrid business solution which relies on the use of games to encourage increased participation and productivity from employees. Productivity Games are designed to increase productivity through the use of gaming elements and engaging game play. Play is part of being human and can help bring people together to have fun, work as a group and accomplish a task^v.

Often, this is done within the context of a game. Stuart Brown’s research into the concept of play highlights the fundamental elements of human play and showcases the essential roles of trust and community^{vi}.

A business process can be viewed as a sequence of activities and tasks that are performed to accomplish a specific organizational goal. As we looked at the characteristics of serious games at work it became apparent that these games were actually variants of business processes. In its August 2008 report, Forrester notes, “The strongest ROI and ultimate adoption will be in serious games that help workers do real work. We are already seeing this with the use of games in product development and collective intelligence, but the real dynamic idea is to pull out the incentive structures and tools of games to boost productivity and employee morale^{vii}”. All of this helped make the case for an increased investment in games.

In a classic statement on the power of working together, Eric Raymond stated in his seminal document *The Cathedral and the Bazaar* that “Given enough eyeballs, all bugs are shallow^{viii}”. While finding software defects is easy when many are involved, the challenge for many tasks is how to motivate group participation. When people get involved in a software beta program or open source project, they have shown an intrinsic interest in participating. However, if they are not involved in efforts like this, other types of motivation are required to encourage participation. We have found that by designing games that incorporate fundamental elements of play, people can be enticed to participate. Even better, if the game play was interesting enough to the players, they would be willing to perform productive tasks in order to participate whether or not they had an intrinsic motivation to accomplish the goal. In our experience and game deployments, this has proven to be true.

In this paper, we will describe in some detail a Productivity Game deployed by the Windows engineering team to address a complex software localization problem that could not have been solved in a cost-effective way without massive participation. Additionally, we will describe briefly additional Productivity

Games deployed to aid with other efforts during the Windows 7 development timeframe. These examples, and the results generated provide a strong case for greater use and research into Productivity Games.

2 BASICS OF PRODUCTIVITY GAMES

Productivity games are related to crowd-sourcing or human computation efforts, but with some key differences. Similar to recognized crowd-sourcing efforts like Wikipedia, or human computation initiatives such as the ESP Game^{ix}, Productivity Games enable employees to have fun participating and feel good about accomplishing productive tasks in the process. The key difference between Productivity Games and crowd-sourcing is the use of gaming concepts to motivate participation in work-related tasks. The evolution of the ESP Game into the Google Image Labeler^x, and the subsequent production of actual business data for Google is an example of a Productivity Game.

Productivity games are not a universal solution for every business process or task. Games introduce an alternative incentive system into the workplace as a byproduct of the game architecture and scoring of play. Since the workplace usually already has an incentive system in place – usually in the form of a paycheck, Productivity Game designers must be careful when, where and how they deploy games that can potentially impact existing incentives and rewards.

2.1 GAME APPLICABILITY

Work tasks draw upon employee skills that can be grouped into one of three categories: core, unique, or expanding. Employees share “core” skills, such as the ability to type, that may be specific to their industry, but do not differentiate employee A from employee B. Some employees have “unique” skills that require specialized training or experience. “Expanding” skills are what employees aspire to and acquire over time to help them perform their jobs better.

From an organizational perspective, there are two categories of tasks that relate to the goals of the organization: “in-role” tasks and “Organizational Citizenship Behaviors” (OCBs)^{xi}. In-role tasks are the tasks that employees are paid to perform. Organizational Citizenship Behaviors are the behaviors that an organization would like employees to voluntarily exhibit to enhance the workplace culture and environment.

From a Productivity Games viewpoint, the employee categorization and the organizational classification overlap in a way that can help identify whether or not a game will be successful in modifying behavior and having people “play”.

Table 1 illustrates the areas where Productivity Games can be the most successful. Focusing either on expanding skills in role, or OCBs that require core skills are the best way to ensure the success of the game. Examples of why specific segments work or don’t work are described below.

	Core	Unique	Expanding
In-Role Behavior			👍
Organizational Citizenship Behavior	👍		

Table 1. Successful Game Deployment

2.1.1 THOUGHT EXAMPLES: WHERE GAMES WORK

Based on our game experiences, described somewhat below, games which encourage good corporate behavior (or OCBs), but rely on core skills that all users share, are the most valuable space for Productivity Games. Since the games rely on core skills, all employees in an organization are able to participate. Additionally, since the behavior is not closely linked to any individual's job, no one's employment is threatened by the success of another team member.

For example, imagine a game that helped sort a complicated list of items. All employees in a given department are familiar with the items, and with how the organization prioritizes its work. This provides a great place for everyone to participate on equal footing. But wrapping the sorting and prioritization work in a game-like interface, all players are given a fair chance to contribute and potentially win.

"Games for Learning" is a well-established genre of software development, and many examples are available in the marketplace for children of all ages. Game work in this space because they focus on the development and growth of the individual. Games are designed to encourage learning, and then test for the learning within the context of play. Players are best rewarded in this space by showing how they have improved themselves, rather than comparing raw completion numbers, which can quickly show disparity between students, but the element of the value of play is not lost.

2.1.2 THOUGHT EXAMPLES: WHERE GAMES DON'T WORK

To further illustrate where Productivity Games can be successful versus less so, let us provide some example scenarios which might better illustrate possible games.

First, imagine a game which encompasses the daily tasks and work of a single employee, Joshua. In the "Joshua Game", which maps to the 'core' and 'unique' skills that Joshua performs for his work, players are given points for doing tasks Joshua would normally do. Some players are able to do all the tasks Joshua is capable of, and some are limited because they do not have the same 'unique' skills that Joshua has.

This presents our first problem: games which exclude players are not in the best interest of the organization. Since Productivity Games rely on a broad number of players, the objective of most games must be to add as many players as possible. Games which rely on actions from the bucket of 'unique' skills inherently limit the breadth of players available to play the game.

Back to the example, we find another challenge. If the end of the "Do Joshua's Job Game" comes and Joshua hasn't won, how does that fit in with his performance review? One thing for certain is that Joshua does not feel secure in his job anymore.

These two issues provide examples why games focused on 'unique' skill sets are difficult to deploy. Additionally, we see how competitive games focused on 'in-role' behaviors can introduce some awkward situations into the workplace and existing performance review processes.

2.2 ENGAGEMENT

One indirect consequence of Productivity Games is the increased engagement of employees in the organization. From literature referenced above we know the "gamer" generation have invested a significant portion of their lives in playing games. And it is interesting to identify some of the attitudes and lessons which this younger generation has taken from playing these games. For example, gamers have learned from games that the cost of failing is very low, and they can always retry, yet from this they expect clear feedback as to what they need to do to change their play in order to succeed later on. From this we can see that the younger generation values a feedback loop and transparency in the consequences. Gamers always expect the game to be fair; otherwise, they will not continue to play. They map these same expectations in a game into their job, expecting the workplace to have transparency and a clear feedback loop. They also expect fairness in how they are treated and in how they should treat others. Finally, games don't demand lengthy reading or studying of manuals in order to play. Most games provide an introductory training mode where the player is given the opportunity to learn what they must know in order to move forward into the game. Similarly, in the workplace, the lengthy corporate memo outlining detailed reasons for organizational priorities carries less impact than is desired.

Productivity Games provide an opportunity for an organization to communicate an organization objective or priority in a method that easily meets the needs of this younger generation. In a properly designed game, fairness and transparency are in place. A feedback loop demonstrating success or failure clearly teaches and trains employees how to change their behavior. And finally, instead of a lengthy manual or memo, an employee has the opportunity to engage quickly and easily in a "training" mode which provides the basic information required for the employee to play the game. This isn't to imply that employees are more apt to receive criticism (constructive or otherwise). Rather, because the "teaching" or "coaching" is framed in a game, they receive the feedback in a manner they are accustomed to learning from already.

With so many of the needs met for younger employees to understand and learn from organizational priorities, productivity is higher, morale is higher and employees' engagement is stronger. We have witnessed this first hand within our test team by monitoring existing productivity metrics (such as average number of defect reports produced each week) and noticing that through several game cycles the metrics stayed constant or improved. This is significant because many games ranging in size and scope were played. Some of the games had output which was additional defect reports, but not all. Though a seeming conflict can be created for employees between their paycheck (primary reward system) and the game (secondary reward system) when the game is sufficiently motivational, the increase in teamwork, morale and engagement are valuable.

3 LANGUAGE QUALITY GAME

The Windows Language Quality Game has been a successful Productivity Game. It addresses organizational citizenship behaviors by calling on employees within Microsoft to apply their core native language skills to help assess the quality of Windows translation efforts.

The traditional business process uses specific language vendors to perform translation work, and then a secondary vendor to assess the quality. The business challenge has been that, for some languages and locales, finding two independent vendors can be difficult and costly. To address this problem, the

Language Quality Game was developed to encourage native speaking populations to do a final qualitative review of the Windows user interface and help identify any remaining language issues. The goal was to ensure a high quality language release and using the diverse population of native language speakers within Microsoft has enabled the pre-release software to be validated in a fun and cost-effective way. The list of Windows languages can be found on Microsoft.com^{xii}. Table 2 illustrates the success that the Language Quality Game achieved as run against interim builds of Windows 7. A more detailed description of gameplay can be found below in a later section, but the goal of the game was to achieve reviews of screenshots and dialogs for translation accuracy and clarity. Native language speakers were encouraged to play from across Microsoft's diverse, international population. The results here demonstrate an immense amount of effort applied to playing the game.

Game Duration	One Month
Total Players	> 4,600
Total Screens Reviewed (Points Earned)	> 530,000
Average Screens per Player	119
Top Player Reviewed	> 9,300
Total Defect Reports	> 6,700

Table 2. Language Quality Game Statistics

Success in the game was defined as the amount of coverage of screens across the 36 languages tested. With the incredible response, most languages had several reviewers provide feedback per screen. Because of the latency in reviewing the feedback, defect reports were not included in players' scores. But, for the Windows International Test Team, defect reports were the most valuable output of the game.

Logistically, the massive amounts of feedback were handled by the international team with tools specially designed to display aggregated feedback. The "Moderator" role was filled on a per-language basis from the ranks of the international team, and allowed the review of multiple pieces of feedback per screen quickly and easily. Where there was obvious consensus from the game players, a defect report would be created. Reviewed screens lacking consensus were quickly reviewed, but at a lower priority and more quickly, such that the screens with the highest likelihood of fixable defects were handled quickly and efficiently.

3.1 BUSINESS PROCESS CHALLENGES

The Windows Language Quality Game provided a solution to challenging business problems that could not be easily solved through traditional processes.

Software development, particularly at the scale of Windows, requires sensitivity towards cultural and political issues. While language issues like this may not impact the reliability of the application, users may react negatively and seek alternatives. In addition, government purchases can also be impacted by mistakes in language translation. As a result of these risks, it is imperative that the Windows Team develops software in a robust way that eliminates cultural and political defects.

The typical process involves finding two vendors; one to do the translation work, and the other to help with quality assessment. As an example, Galician is the language of Galicia, in Northwestern Spain. Portuguese speakers can understand Galician and sometimes refer to it as a dialect of Portuguese. However, there are cultural and dialectal differences that must be accounted for specifically in the Galician version of Windows.

Translation or geopolitical errors can impact the quality, perception, and sales for a region. In Windows XP, for example, users can set up a profile by entering details such as their age, sex and number of children. A version distributed in Latin America asked users their gender, giving their options as No especificado (unspecified), varon (male) or hembra (female). Unfortunately, in some Latin American countries the term hembra has a negative connotation^{xiii}. As a result, additional care must be taken to ensure that localized versions of Windows can be distributed to all countries where that language may be spoken.

3.2 GAME ARCHITECTURE

The Language Quality Game is built using a SQL Server database of images that are rendered in the game using Silverlight. The Windows International Team uses an automated process to copy dialog images from the Windows source code into the SQL server database. The dialogs are then augmented with metadata about the language and usage of the image in question.

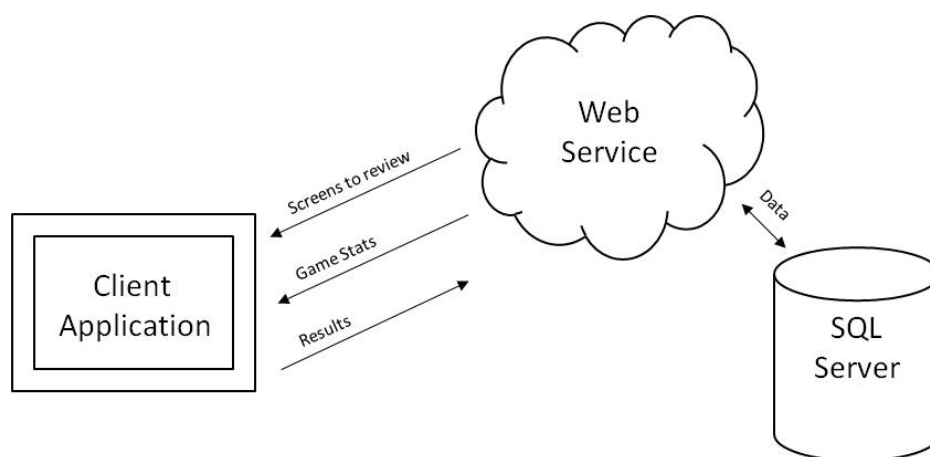


Figure 1 - Language Quality Game Architecture

The dialog images are broken up randomly into groups of 25 to provide multiple “levels” for the player to achieve. As players work their way through the game, each dialog is presented. Players can use their mouse or a digital pen to circle errors using electronic ink. Ink feedback is stored efficiently along with the dialog ID in error reports. This not only saves space in the database, but it also improves performance and helps with results reporting.

3.3 PLAYER POPULATION SELECTION

Finding players to perform the human computation work of reviewing dialogs in the Language Quality Game can be a challenge. It is critical to find native speakers for all the languages supported by Windows versions. For the Language Quality Game, players were recruited by sending broadcast email announcements to native language speaker social aliases, or email distribution lists available internally at

Microsoft. Invitations were sent via email to groups such as “Persian Speakers at Microsoft”, asking members to visit the Language Quality Game web site and play the game.

Finding the right aliases of potential players was critical to the response rate. We also found that native language speakers typically have friends and relatives who will be using localized copies of Windows. Therefore, it is in the speaker’s best interest to play the game and help ensure the quality of the localized version that is important to them.

3.4 DATA QUALITY AND CHEATING

While it’s not possible to completely prevent cheating in a way that scales and keeps people actively participating, it is possible to inject “known defects” and ensure that players find and record them. This helps assess the reliability and validity of an individual player’s answers and allows for filtering. In addition, for the Language Quality Game, there is an assumption that players’ personal desire to improve product quality for their own native language outweighs the desire to cheat. This is furthered by producing a game where no prizes were offered. Leaderboards within the game certainly provided some motivation and competition among players, but between national pride and the limited value of prizes, we believe the incentive to cheat was minimized. Further study is certainly warranted to understand whether successful participation in these kinds of Productivity Games influences annual reviews, etc.

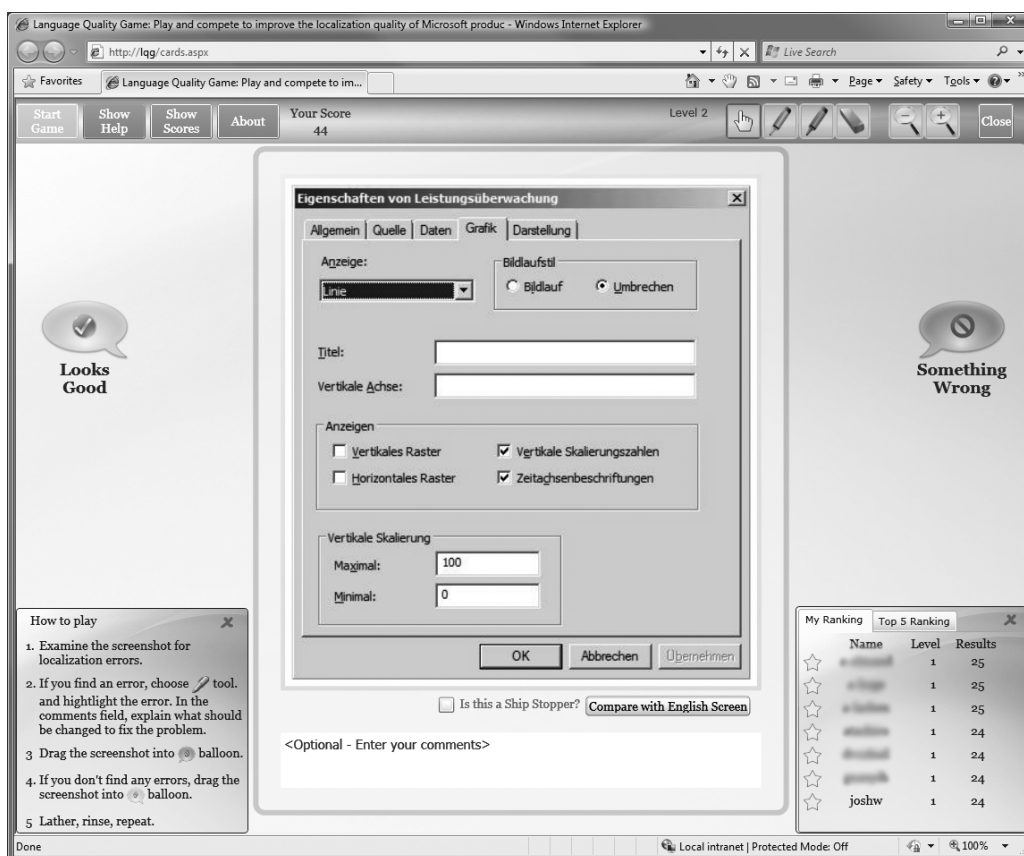


Figure 2 - Language Quality Game Screen Shot

3.5 FEEDBACK LOOP

In order to allow users some knowledge about their contribution to product quality, a report was provided that displayed their scores on a per language basis, but also provided a count of defect reports filed based on screens they had reviewed. Additionally, they were also provided a count of bugs filed on screens which they had reviewed as “good”. This dual-feedback didn’t cover every logical possibility for the combination of outcomes, but did provide users a simple method of knowing whether they were being sufficiently critical in their feedback or not. This kind of feedback adds to the experience of the player, as they are able to learn from feedback about their own performance.

3.6 GAME ELEMENTS

While the language quality screen review work is not tremendously difficult for native language speakers, it is also not the most interesting or engaging, particularly with a large volume of screens. Consequently, game elements and enticing game play were designed and used to attract players and help motivate them to “play”. These are the characteristics of Productivity Games that help differentiate them from other crowd-sourcing efforts.

3.6.1 GAME LEVELS

The dialogs are broken up into groups of 25 images and presented as “game levels”. Once players review all the images in one level they move to the next higher level and are presented with a new set of 25 images.

3.6.2 EARN MARKUP PEN COLORS

There are multiple markup pen colors. As a player reviews more and more dialogs, they can earn and use a different color pen.

3.6.3 GRAPHICAL IMAGE MOVEMENT

After a player marks up a dialog, they move it to either the “Looks Good” or “Something Wrong” pile. This movement and displaying the next dialog involve some basic Silverlight animation which adds visual interest and a gaming feel to the experience.

3.6.4 LEADERBOARD

Each person can view a leaderboard showing all players, their current game level and how many dialogs they have reviewed. Not only does this allow each person to assess their relative effort, but it also provides the basis for some friendly competition. The leaderboard is divided up in a variety of categories – by language for instance – to encourage participation.

3.7 LONGEVITY

Like many games, Productivity Games have a limited lifespan for the work that needs to be done. But, in addition, there is a risk of burnout among the players in doing the task involved in the game. It is not safe to assume infinite play from all games. So, as a method to rejuvenate participation in the Language

Quality Game, a set of new screens was provided to the game players during week three of play. These new screens were the result of early defect reports and fixes provided during the game. With the new screens, a new series of announcements via email were released to inform players that their feedback had been heard, and now we needed their participation again to help review the repaired screens. This did drive a second surge of participation.

Most Productivity Games can benefit from this same strategy. As the game progresses, there are cases where the priorities of the organization have changed, or the rules of your game have been taken advantage of to the benefit of one or few players. Sometimes “resetting” the game with amended rules or providing new content can help reinvigorate players and bring additional life to your game.

3.8 LANGUAGE QUALITY GAME RESULTS

There has been 100% language participation – all 36 languages have been sent out for linguistic review and reviews have been received for all of them. The participation ranged from Korean, with over 82,000 screen reviews to Finnish, with under 1,000. Across the board, over 7,000 defects were reported across all the 36 languages.

After validation and data quality assessment, an average of 85% dialogs were found to be completely correct – the highest was Slovakian with 92% of screens reviewed marked as correct and the lowest was Bulgarian with 65% of screens marked as correct.

At the end of the four-week play period, there were over 4,600 players. The language with the most had 615 players and the least had 10 players.

4 OTHER PRODUCTIVITY GAMES

Microsoft has also tried other styles of Productivity Games in a variety of forms and sizes over the past few years. The games with the greatest participation were the games used in the Windows Vista Beta program. This experience is covered extensively in chapter 5 of “The Practical Guide to Defect Prevention^{xiv}”.

More recently, a Productivity Game was created and used to classify freeform text comments as “actionable” or “not actionable”. This feedback was generated by beta testers of Windows 7, and returned to Microsoft using a built-in tool which gathered this kind of text-based comments and feedback. Traditionally, this feedback categorization has been performed manually by the software team and is time-consuming and labor intensive. In some cases, automated machine translation and “text-crunching” tools have been tried with limited success, and still required a human step for final validation.

The strong interest in college basketball tournaments was used to attract potential players. The Feedback Productivity Game was structured as three phases, one before the tournament started and the other two phases related to subsequent rounds. The goal was to keep the duration of each “sub-game” short, vary the format slightly, and keep interest levels in the game high.

To participate in the Feedback Productivity Game, the player had to gain credits by classifying text comments into “actionable” or “not actionable”. A screen was displayed with the ability for users to categorize each comment. For each comment classified, one game play credit was received. The credits could later be used in each round to play different “games”.

The pre-tournament phase provided each player with a random pair of basketball teams (from the 64) and they could then select the one they thought would win between this hypothetical pairing. Players made selections by clicking on the name and logo of the school they preferred. Each selection required one comment classification credit, and immediately another choice was placed before them. Once credits were consumed, the player was again encouraged to classify additional text comments.

The next phase of the Productivity Game mapped to the teams who remained in the playoffs, and had real matchups displayed. The player could then select who they thought would win in that matchup. Each selection again required one comment classification credit.

The final phase of the game focused on the final teams in the tournament. To play, each player exchanged four credits for a “team ticket” indicating that team would win it all. Multiple tickets could be obtained for each team and tickets could be obtained for multiple teams. The objective of the game was to obtain tickets for the team that actually won. All players with tickets for the winning team would earn points in proportional to the number of tickets they had.

A total of 150 players participated in classifying 4723 feedback comments and 53% were assessed to be “actionable”. These results saved the Windows team a tremendous amount of effort by distributing the work across basketball fans with these core skills.

This Productivity Game differed somewhat from the Language Quality Game where the relationship between play and work was more unified. But, the motivational factors were similar in that play of a game (or in this case, the ability to make my picks for games) was enabled by accomplishing a task useful and valuable to the organization.

5 CONCLUSION

In this day and age, many business challenges can benefit from groups of people working together to provide solutions. Recently, crowd-sourcing has been used to distribute tasks that can benefit from human computation. This same concept can be utilized in corporations to tackle tasks that they are not resourced to support or that require unique skills such as native language proficiency.

A challenge in any of these efforts is how to entice and motivate people to participate. The Productivity Game concept utilizes gaming elements and engaging game play to help generate that motivation. Through Productivity Games like the Language Quality and Feedback games, we have shown that people can become engaged in a game and willing to exchange “real work” in order to participate. These results have demonstrated to us the tremendous potential of Productivity Games to help solve problems that are difficult or impossible to solve within traditional organizations and business processes. We look forward to the continued pursuit of that potential.

6 AUTHOR BIOGRAPHIES

Joshua Williams is a senior software quality engineer at Microsoft, currently working on the Windows Core Security Test Team. Over the past 14 years, he has contributed to the development, testing and shipping of Windows Operating Systems. Across at least nine different releases, Joshua has been an individual contributor, lead and manager across a variety of component teams and technologies. These include the Windows International Team, focusing on versions of Windows besides English and the USB and 1394 teams, shipping support for numerous external device classes, including the first USB 2.0 (Hi-Speed) devices. More recently, Joshua has focused on large-scale test automation frameworks and development lifecycle tools to improve test and development productivity. As part of the Windows Security Test Team, Joshua has researched new ways to engage team members, working to improve the quality of work, but also the quality of life of the team. Joshua loves all kinds of games, and brings that passion to his work on “Productivity Games”.

Ross Smith has been in the software industry for over 20 years, developing and testing software on everything from mainframe systems to handheld devices and PC's. He began his Microsoft career in Product Support in 1991 and has been a Test Lead, Test Manager and Test Architect. He has been a long-time member of the Test Architect's Group, and has worked on almost every version of Windows and Office since 1995. He is one of the authors of “The Practical Guide to Defect Prevention” and holds 5 software patents. Over the last couple years, he has nurtured a management innovation initiative called 42Projects (www.42projects.org), aspiring to inject cultural change by using trust and games to make work even better at Microsoft.

Dan Bean has been in the computer industry for more than more than 27 years and with Microsoft since 1993. Like many people with long careers in the computer field, Dan has had the opportunity to work in a variety of disciplines including systems design, program management, development, test and information technology. As a member of the Microsoft Engineering Excellence Group, Dan worked on engineering practices to improve software quality. In addition to earning a Computer Science degree from Washington State University, Dan also earned a Black Belt in Six Sigma from the Juran Institute.

Robin Moeur has a professional career story that spans more than 30 years. Fascinated by the problems that arise in corporate hallways and the cubicles workers inhabit around the world, she has developed significant insight into work processes, resistance to change and what it takes to find success through creative solution deployment and management innovation. After eight years in professional services, Robin accepted a position at Microsoft contributing to several ‘firsts’ on the Redmond campus, including the launch of one of the first two live content sites created after the launch of MSN in 1995, known then as //sidewalk. She continued to grow her career and leverage her experiences as a program manager and product planner, crossing paths with Ross where they partnered together on a product team. Robin was invited to fill a temporary position as Adjunct Professor in the School of Business at the University of Montana in 2001, filling a lifelong dream to leverage her years of learning in business to excite undergraduate and graduate students on their unlimited potential to impact the world. Returning to Microsoft in 2003, she managed a team focused on security of the global network. For the last several years she has continued pursuing her passions, as a consultant, author and research partner with Ross focusing on organizational behavior, corporate culture, change and management innovation. She is actively collaborating with faculty at the University of Washington creating leading edge service learning curriculum; Web 2.0 projects aimed at post disaster economic recovery, connected communities and civic engagement; and, is an advisor/guest lecturer in the Masters of Digital Media program.

7 ACKNOWLEDGEMENTS

Our thanks to the following who continue to be instrumental in the success of Productivity Games:

- Darren Muir
- Harry Emil
- Robert Musson
- James Rodrigues
- Karen Djoury
- Jian Chen
- Delton Porter

ⁱ The Serious Games Initiative, <http://www.seriousgames.org>

ⁱⁱ Choonghoon Kim, Douglas Michele Turco (1999). "The Next Generation in Sport: Y", Cyber-Journal of Sport Marketing, <http://fulltext.ausport.gov.au/fulltext/1999/cjism/v3n4/lim34.htm>

ⁱⁱⁱ Wikipedia, http://en.wikipedia.org/wiki/Generation_gap

^{iv} John C. Beck, Mitchell Wade (2004). Got Game: How the Gamer Generation is Reshaping Business Forever. Harvard Business School Press. ISBN 1-57851-949-7

^v Michael Elliott (2008), "The Games that Bring Us Together", Time Magazine, http://www.time.com/time/specials/2007/article/0,28804,1815747_1815707_1815705,00.html

^{vi} National Institute for Play: www.nifplay.org

^{vii} T.J. Keitt, Paul Jackson (August, 2008) "It's Time to take Games Seriously", Forrester

^{viii} Eric S. Raymond (1999, 2001). The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly. ISBN 1-56592-724-9.

^{ix} Wikipedia, http://en.wikipedia.org/wiki/ESP_Game

^x Google's Image Labeler, <http://images.google.com/imagelabeler/>

^{xi} Wikipedia, http://en.wikipedia.org/wiki/Organizational_citizenship_behavior

^{xii} Microsoft Help and Support: Knowledge Base, "List of languages supported in Windows 2000, Windows XP and Windows Server 2003", <http://support.microsoft.com/kb/292246>

^{xiii} Nic Fleming (2004), "It's a tricky world in computers, says Microsoft Chief", Telegraph, <http://www.telegraph.co.uk/news/worldnews/1469733/Its-a-tricky-world-in-computers-says-Microsoft-chief.html>

^{xiv} Marc McDonald, Robert Musson and Ross Smith (2008). The Practical Guide to Defect Prevention. Microsoft Press. ISBN-13: 978-0-7356-2253-1. www.defectprevention.org or <http://productivitygames.blogspot.com>

New Challenges to Quality in the 24x7 Enterprise I.T. Shop: Post-Integrated Business Automation Systems

Al Hooton, CIO
al@hootons.org

August 2009

The traditional 24x7 IT operations organization is often perceived as only needing sub-par technical professionals. This has often not only been the perception but also the reality for many of these organizations because the systems they would purchase and deploy have been “pre-integrated” by the companies providing the systems. All too often this limited the ability for the IT operations professionals to get very far under the hood, relegating them to roles requiring less technical prowess than their product development counterparts. Reality has been changing rapidly in the last several years as new approaches to the design and integration of complex business systems has thrust these operations professionals into a new world of “post-integrated” systems. This paper discusses the four primary forces behind these changes, and provides a few examples of approaches the 24x7 IT shop can borrow from their product development brethren in order to meet the skillset and innovation requirements of this new post-integrated age of business systems.

Al Hooton has been professionally active in the software development and information technology industries for the last 30 years, holding various positions including computer operator, hardware/software engineer, chief architect, quality assurance manager, Director of Technical Operations, Chief Information Officer and CEO/President. His experience is evenly split between software product development positions and 24x7 I.T. operations positions. Mr. Hooton holds a Baccalaureate degree in Computer Science from Purdue University and a Master of Science degree in Computer Science from the University of Louisiana.

Copyright © 2009, Al Hooton. All Rights Reserved.

Published at the Pacific Northwest Software Quality Conference 2009

Traditional Differences Between Computer Engineering and Computer Operations Personnel

For most of the 65+ years that electronic information processing systems have been in use, there has been a distinct difference in perspective regarding the skills necessary for “designing and building stuff” using computers versus “running stuff” using computers. Designing and Building Stuff (otherwise known as one or more areas of *computer engineering*) requires academic foundations, formal techniques, ascendancy through multiple levels of mentored guidance provided by more senior engineers, and various other experiences that result in one possessing a highly respected set of skills. On the other hand, Running Stuff (otherwise known as *information technology operations*, or “I.T.”) is perceived to require few or none of these things, being entirely addressable by smart self-learners who have taught themselves how to fix password problems and call the vendors (or engineers) who created the systems they are running when anything difficult has to be done.

I.T. professionals are understandably quick to defend their abilities, and will argue strenuously that they are expected to have the same levels of competence as their engineering counterparts. When the author has been thrust into debates regarding this set of assertions, the argument is quickly resolved by retrieving several current job postings for both engineering and operations positions for comparison. Here are excerpts from two job postings current at the time this paper is being written, both of which are looking for “mid-level” professionals:

Computer Systems Engineer

- Bachelors degree in computer science, computer engineering or electrical engineering required; Masters degree highly preferred
- 7-10 years experience designing, building and maintaining complex information systems
- High levels of proficiency with the following technologies: UNIX/Linux, Java, EJB, SQL, C/C++, operating system internals
- Prior demonstrated success working as part of an engineering team

I.T. Operations Engineer

- Bachelors degree desired; high school diploma with equivalent experience may be substituted
- 2 years experience in the operation and maintenance of both Windows and Linux systems
- Familiarity with concepts and usage of the following types of systems: ERP, accounting, human resources, data warehousing
- Familiarity with shell scripting and writing administrative tools in perl

This is not an indictment of the operations professionals themselves, rather it simply points out there are different expectations in the formal education and technical experience levels on the part of the information systems industry overall. There are certainly exceptions to this rule, but anybody who is active in this industry will recognize the differences as more common than not. The operations professionals are often viewed as “second-string” resources who do not need to be as technically competent as their engineering counterparts.

Historical Causes for These Differences

The differences in technical skill level between engineering personnel and operations personnel is most often real, not just perceived. Businesses are perfectly happy for these differences to exist because operations personnel are typically paid less than engineering personnel. The accountants and executives see this as an advantage since the computers keep running almost all the time anyway, at lower labor cost.

This state of affairs has developed over several decades. While there are several contributing factors, the author contends that the most influential is the “pre-integrated” nature of typical enterprise-scale information systems.

Successful I.T. shops have brought in business analysts to work with all the groups around the company to understand the relevant business requirements, and would then evaluate large systems from several vendors who each claimed their system could meet the requirements. A choice would be made to select a single vendor for software and a single vendor for hardware (often the same) because this was the only way to ensure all the pieces of the system worked together [1]. These traditional systems lack flexible integration points at their edges and typically utilize proprietary internal integration architectures between their primary functional subsystems, thus are referred to here as being pre-integrated.

These pre-integrated systems typically go through an extended deployment period, during which the vendor(s) provide costly professional services to make changes to the selected system so it actually does meet the needs of the business. The operations personnel would be unable to make these changes even if they had all the correct skills, due to the closed nature of these systems and the lack of external integration interfaces. For these same reasons, once the system is deployed the operations personnel are typically relegated to a role of monitoring the system to ensure it is functioning correctly and calling the vendor for help at the first sign of trouble or when functional enhancements are required.

An obvious issue with these approaches is the Total Cost of Ownership (TCO) for such pre-integrated systems. Any experienced enterprise I.T. executive will tell you this scenario results in the on-going costs for a system far outweighing the initial costs of purchase and deployment. Vendors of these systems know they essentially have their customers stuck with no alternatives but to come back for help or modifications, so those services become very expensive. Until recently this was the path most likely to lead to success for the operations personnel in supporting the needs of the business, regardless of cost, so it was the one most often chosen. However, in the last several years, some fundamental changes in software technology have altered the I.T. operations landscape. These changes are not only driving fundamental shifts in how I.T. shops purchase, build and deploy information systems but they are also driving changes in the skills that operations personnel must possess in order to succeed.

The Dawn of “Post-Integrated” Enterprise I.T. Environments

The realities described above have been shifting in the recent past, with the result that everyone in the enterprise I.T. shop is struggling with new expectations, new technical challenges and a completely different approach to the purchase/build/maintain cycle for the business systems they deploy. The primary reasons for this shift are described in the next several sections of this paper, but the combined result may be described as a move away from “pre-integrated” systems to “post-integrated” systems. This new world is one where the overall set of functionality deployed to meet the needs of the business is no longer procured from a single vendor. Rather, there are systems from many vendors being procured, modified, integrated and deployed, in addition to systems that may be constructed from scratch internal to the I.T. shop. This is a radical departure from the past, and is driving operations personnel to be proficient in enterprise-scale software development, multi-level systems integration, management of long-term architectural roadmaps that drive system evolution, etc. In short, operations personnel now must be proficient in all the skills previously expected only of computer engineers.

The primary reasons we are entering the post-integration era are discussed in the following sections of this paper. In one way or another they all provide *new ways to think about and implement integration between heterogeneous systems that is driven by business requirements instead of technical constraints.*

Service-Oriented Architectures (SOAs)

With traditional enterprise-scale information systems, it was effectively impossible to utilize functional primitives within the system except through a user interface provided by the vendor of the system. For example, a warehouse management system might implement a functional primitive such as “move N items of part number X from the overstock area to the manufacturing area”, and expose that functional primitive through a series of user interface screens on terminals in the warehouse. However, there were no other interfaces that could be used to trigger the execution of this functional primitive. Thus, even though a business process might exist that requires the value of the moved inventory to be credited to an overstock financial account and debited from a manufacturing financial account, there is no way to integrate the accounting system with the warehouse management system in order to automate the enforcement of the business rule.

SOAs [2, 3] allow systems to be built such that functional primitives communicate with each other, and with

external systems, via standard mechanisms called a “service interface” or “web service” (depending on technical details). By using such a standard mechanism for both internal integration and external integration, functional primitives from across multiple systems (i.e., the warehouse management and accounting systems) may be combined in order to automate cross-functional business processes. Figure 1 shows an example enterprise architecture where a workflow engine is programmed to implement business processes by utilizing functional

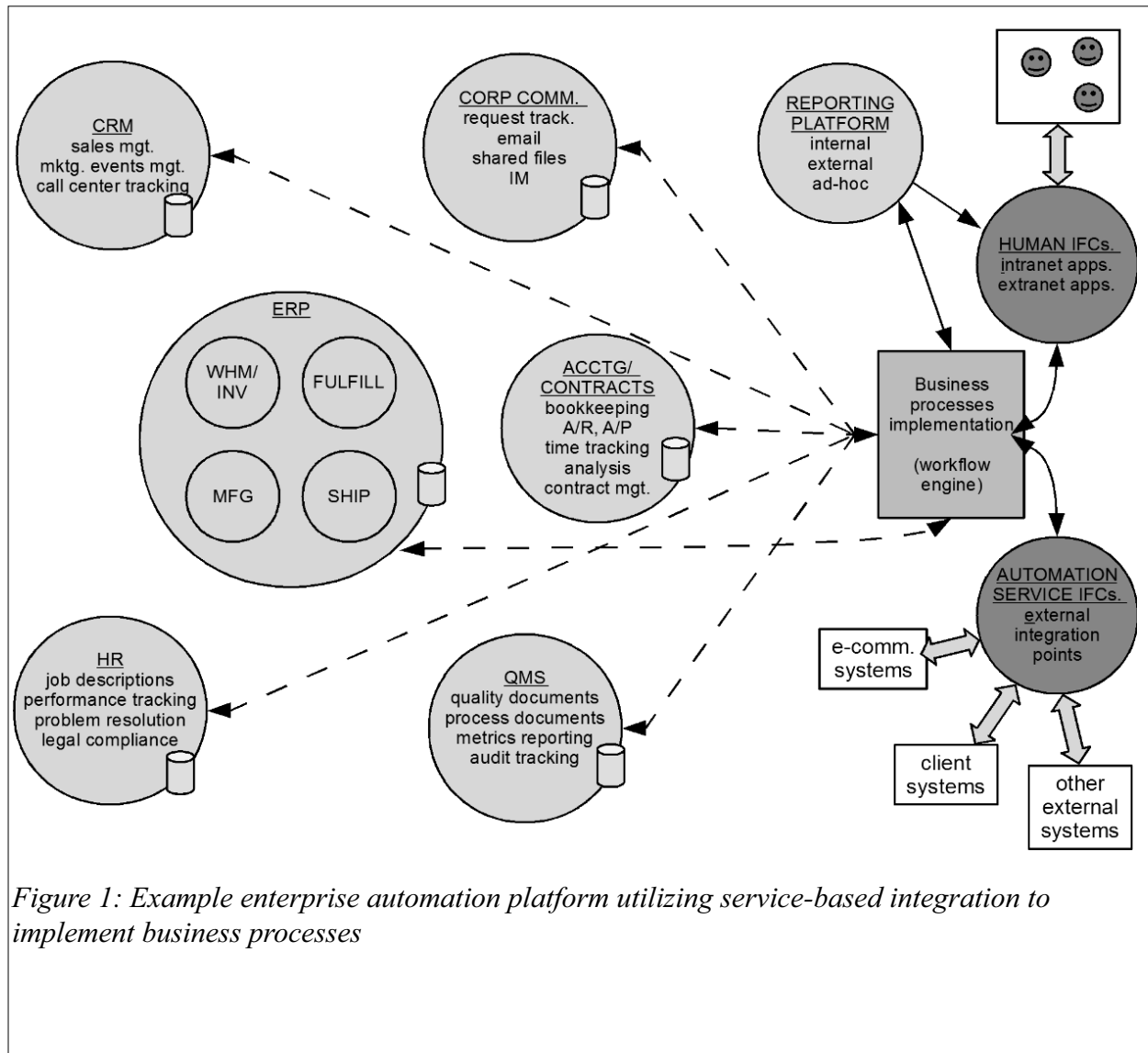


Figure 1: Example enterprise automation platform utilizing service-based integration to implement business processes

primitives across all the back-end systems.

Enterprise-scale SOA approaches allow multiple systems from disparate vendors to be easily integrated by parties other than the vendors. This work is most often done in the I.T. shop by the operations professionals, requiring them to be proficient in many areas of software development, integration and testing.

Core Services versus Non-Core Services

The traditional enterprise I.T. shop runs all automation systems necessary to support the business on-site, or “behind the firewall”. This would include the most critical systems, such as the accounting system and systems vital to

generating revenue, as well as non-critical or niche systems such as email list servers or limited-use websites. Since everything was pre-integrated this was often required, but it resulted in the operations personnel being diverted from ensuring the critical systems were running whenever a non-critical system needed attention because everything was internal and running on the same infrastructure.

In recent years a philosophy has emerged that suggests the I.T. shop does not provide *systems* to the business, it provides *services* [4]. Those services are automated on computer systems, but there may or may not be a one-to-one correspondence. For instance, the business may require automated inventory tracking services in order to be competitive. Internal to the I.T. shop there may be multiple systems that are used together in order to provide these services. This philosophy can provide the I.T. shop a great deal of flexibility in how it plans and deploys systems, but only if the commitments it makes to the business are truly made in terms of services instead of systems.

Once the commitments are defined in terms of services, the I.T. shop has the ability to work with the business to define *core services* and *non-core services*. The definitions will vary between organizations, but in general terms the core services are usually directly related to the immediate generation of revenue and require high availability, high data reliability and high visibility. Alternatively, the non-core services are usually not directly related to the immediate generation of revenue and can operate at lower levels of availability, etc. This allows the I.T. shop to make different implementation and priority choices when determining how to deliver services to the business, and allows operations personnel to more effectively spend their time on systems that deliver core services.

This set of philosophies can significantly alter the demands on operations personnel if it is determined that systems delivering non-core services can or should be physically disparate from those delivering core services. This is a frequent outcome, due to the high cost of hosting systems in operational environments that provide everything required to meet the availability and reliability needs of core services. As soon as separate physical locations are involved between these two sets of systems, operations personnel will be required to master sophisticated network engineering, replication, fail-over and remote management skills.

Software as a Service (SaaS) delivery mechanisms

In recent years the model for delivering non-core services to the business has started to shift toward the utilization of Software as a Service (SaaS) delivery from a mix of external providers. SaaS [5] allows delivery of services to the business from remote data centers on a subscription basis, with the business having no ownership of the systems delivering those services or responsibility for maintaining those systems. There are several established examples of this, such as Salesforce.com, as well as many emerging examples providing a full range of business services. In his current position, the author has directed several initiatives that have resulted in many non-core services being migrated or launched on systems that are delivering the services in a SaaS model, including all corporate email services, shared calendaring services, company-wide issue/request tracking services, document control/tracking services and corporate intranet services. The result has been increased ability for operations personnel to monitor, maintain and enhance the systems remaining behind the firewall that deliver core services to the business.

Many companies that deliver their solutions using a SaaS model have consciously built sophisticated integration interfaces to these services. The emerging approach for these integration interfaces is to use SOAs and expose the resulting service interfaces through secure networking connections. In this way, I.T. shops that are utilizing capabilities being delivered to them via SaaS mechanisms have the same ability to integrate with those systems as they would have if they were running everything locally.

The emergence of SaaS as an enterprise-ready option further demands that operations personnel become familiar with sophisticated approaches to network engineering, heterogeneous systems integration and assembly of end-user services from multiple functional primitives.

Free Open-Source Software (FOSS)

Although Free Open-Source Software (FOSS) [6] has been around since the 1970's, it has traditionally been viewed with a combination of suspicion and skepticism by enterprise I.T. shops. There are a variety of reasons for this, but the most common is the belief that "there's nobody to call when it breaks". For good reason, operations personnel are most often judged on the predictability of the services they provide – the fastest way for an I.T. shop to get in trouble is when system issues cause the services they deliver to be unpredictable in some manner. The best way to avoid this problem was to ensure there was a vendor who could be called on (and blamed, when necessary).

As a general rule FOSS systems have no vendor, or the vendor is releasing a FOSS version of their system with no included support, as a way to introduce the market to their product and hopefully convince people to purchase a full commercial version. Most FOSS systems are built and maintained by a loosely-connected set of developers who may not have ever met in person and do all their work on the system virtually. While this may give traditional operations personnel reason to pause, there are many well-known FOSS systems that are more stable, exhibit fewer bugs, drive a lower TCO and are repaired much more quickly when problems occur than their commercial competitors. For these reasons FOSS systems are increasingly chosen in many I.T. shops to provide a variety of services to the business.

The appearance of FOSS requires operations personnel to not only become competent with the full range of integration skills described previously, but in many cases will require them to also become fully proficient software engineers. This is required when they must make modifications of a FOSS systems to meet their specific business needs, or they are required to make fixes to the system if a bug appears. FOSS systems allow a number of advantages over commercial systems, but operations personnel must have the skills necessary to successfully make use of these advantages.

Leveraging Well-Known Software Engineering Quality Techniques in the I.T. Shop

With the 24x7 I.T. shop being bombarded with these new challenges (and opportunities) it is common to find operations professionals struggling to successfully meet the needs of the businesses they support. The new opportunities provided by the changes discussed above are not fully leveraged in most shops, and are not being used to any degree at all in many. Operations professionals are in a tough spot, and the only way out will be to aggressively pursue new skills and then apply those skills to the design, procurement, integration, deployment, maintenance and operations of their systems going forward.

Like it or not, operations professionals and also the businesses that employ them need to recognize that fundamental preparation in an academic setting is now becoming necessary for the I.T. shop just as much as it is necessary for product developers. College degrees in Computer Engineering, Computer Science or related fields will become mandatory for success in the era of post-integrated systems. The businesses that employ professionals with these new levels of preparation will need to pay them at levels commensurate with product developers as well. This additional expense will be returned to the businesses many-fold in the form of enhanced capability, flexibility and reliability of the resulting business automation systems.

Through the course of formal education, operations professionals will be exposed to many approaches and techniques that have been familiar to software product developers for decades. Most, if not all, of these techniques are applicable to the world of post-integrated business automation systems. Below are a sampling of lessons derived from the experience of the author in applying well-known software engineering approaches to I.T. operations teams.

Lesson 1: Incremental development, integration and deployment (avoid the Big Bang)

The typical approach to deploying enterprise-scale business automation systems looks a lot like the classic waterfall approach to product development that was discarded by product developers decades ago. In the 24x7 I.T. shop there are often very long periods of requirements gathering and sign-off, followed by equally long periods of defining functional specifications based on those requirements. When all this work is done vendors receive Request For Proposal (RFP) documents that are often hundreds of pages long, and they are expected to respond to these RFP documents with proposals that meet all the needs of the business in a single system to be rolled out across the company in a single deployment. After further long periods of selecting a vendor, having the vendor modify their system to meet the needs, and a long period of testing with a small group of people, the day comes to “flip the switch” to the new system. Product developers often refer to this kind of approach as the “big bang” technique, in that everything is supposed to be fully compliant with the initial requirements all at once, when the switch is flipped. This rarely works, resulting in alternate definitions of “big bang” that are much more negative in nature. This kind of approach is, at best, problematic with pre-integrated systems. It is significantly more difficult with post-integrated systems, and it misses a key potential advantage that post-integrated systems offer: the opportunity to build and deploy in phases, allowing the management of complexity to be tackled in smaller chunks.

Over the last two decades there has been great progress in the product development community in the area of *incremental development and integration* techniques. This started with the work of Barry Boehm in the 1980's [7, 8]

and exists today in various forms that include a set of approaches collectively referred to as *agile development* [9]. All of these approaches share a few important understandings:

- It is not possible to know everything a complex business automation system will need to do before it is first built.
- It is not possible to manage all the complexities of a full enterprise-scale business automation system at the same time. Success comes from dividing the system down into pieces that are incrementally developed, integrated and deployed at different times, some earlier and some later.
- By explicitly recognizing at the outset that there will be many outstanding questions, and utilizing an approach that develops, integrates and deploys the entire system over many incremental cycles, it allows the team to focus on giving the business users what is known to be useful and then leveraging those things to further define what is not yet known before attempting to build, integrate and deploy additional functionality.

24x7 I.T. shops can deliver great value to their business users, more quickly, and avoid suffering one of the often-reported large-scale system deployment failures if they adopt incremental development and integration techniques.

In a recent position, the author was responsible for several groups of professionals in both internal development and operations support roles who were building, integrating and running a large real-time financial transactions platform. The teams had been attempting to utilize a variety of agile approaches, but had been quite disappointed in the results because they found themselves significantly over-committing for each build/test/deploy iteration and subsequently failing to achieve 60% to 70% of those commitments. When the author joined the organization, many of the business people internal to the company were prepared to force the technical teams to abandon their “new” agile approaches because these approaches had failed to meet the needs of the business any better than the previous “big-bang” approaches.

After meeting with each of the teams involved individually, and then through a series of cross-functional meetings with all the teams, a critical lesson was uncovered. Even though many aspects of current-day agile techniques were being used, a fundamental requirement for success was missing: the requirement that each iteration culminate in a customer-ready system. This had been the intent when the switch to agile approaches was first adopted, but the operations team found that customers were unnerved by frequent updates to their production financial systems. Due to this the operations team had made a well-founded decision to “skip” several iterations at a time that were released from the developers. While this is a fine approach, it caused the unintended result of QA and testing efforts to be reduced for the skipped iterations. In this organization, responsibility for the QA and testing work fell to operations instead of development, but when the demands for quality dropped for skipped iterations there were serious defects remaining that went undetected.

Everybody thought things were going great until work started on an iteration that was intended for external release. The QA and testing efforts ramped back up and proceeded to find that the last several iterations had not actually resulted in a customer-ready system being produced. This smaller version of the “big-bang” approach would then cause significant commitment and schedule overruns for the customer-visible releases.

The Lesson: Just because the organization determines that it is not going to externally release the output of every iteration in an agile approach you still must truly produce a customer-ready system with each iteration.

Lesson 2: Integration is the other 90% of the work, focus on it from the beginning

A statement frequently heard on product development teams is that “writing the code is the first 90% of the work, integrating the code into a system is the other 90% of the work.” This is fine when it is the development team that is doing the integration but, as discussed previously, it is increasingly the responsibility of the operations team to do the majority of integration work between heterogeneous systems. The act of integrating complex system components together to form a complete user-ready system is difficult because it is during integration that the management of full system-wide complexity is truly required. This made that much more difficult when the systems being integrated are from multiple vendors, and the operations team was not involved in the original development of most or all of them.

Many years ago, researchers in product development techniques identified integration as the place that the most difficult or complex system requirements would either be met or fail to be met. Two important categories of techniques have emerged from this set of understanding that are in wide-spread use among product developers:

- Sophisticated configuration management approaches to track subsystem dependencies, manage multiple released versions, allow easy and rapid roll-back of any given release if problems emerge, and manage how the overall system evolves through time due to incremental development/deployment.
- Utilization of higher-level modeling techniques to manage evolving system complexity. There are a variety of diagram-based modeling techniques such as Unified Modeling Language (UML) [10] that can be used with both technical and non-technical stakeholders during the early stages of projects to understand how business requirements interact to drive technical requirements.

Several years ago, the author was responsible for both the product development team and the 24x7 operations team at a company delivering a web-based complex document preparation system delivered in a SaaS model. At the time he joined the company, there had been a series of accidental configuration, deployment and operations mistakes that disrupted service to customers. While these mistakes were usually identified and rectified quickly, they had a negative impact on the reputation of the company with both customers and prospects.

In this case the root cause was easy to discover. Even though the development team was keeping comprehensive configuration management records during development and testing of the system, no such efforts were being made to track customer-specific configurations at the time of deployment. This caused two problems:

- The development team was unable to readily see what was happening to the system when it was being deployed, preventing them from understanding how to better build future versions of the system to ease operational problems.
- There was no way to apply automation to either maintaining or re-applying customer-specific configurations after an upgrade of the core system. Manual records were kept, but it was error-prone to re-apply the configurations after an upgrade and there was no way to support automated regression testing against anything except the core system configuration.

The development and operations teams worked to extend the configuration tracking system used in development so it would hold configuration profiles, deployment scripts and automated tools that would compare any deployment of the system against the records in the configuration tracking system to identify inconsistencies. As the first version of this new approach became available, customer interruptions due to incorrect deployment/configuration exercises essentially disappeared.

The Lesson: Managing system-wide complexity only at the time of integration is difficult. By managing complexity consistently (in this case with a configuration management system that spanned the entire life-cycle of each product iteration) it is possible to keep the complexity under control.

Lesson 3: Design it like you plan to deploy it

One of the most common refrains in a 24x7 operations shop is “what were these people thinking when they built the system like this?” This is often heard when the operations people find that something they think about every day of their professional lives, such as being able to monitor the system resources being utilized by an application system, never crossed the minds of the people who built the system. This is a frequent reality, especially in the pre-integrated world of the past, but is also found in the post-integrated world that operations professionals are increasingly supporting.

The good news is that the same forces causing the 24x7 I.T. shop to achieve competency in complex systems integrations also provide many of the hooks necessary to make the resulting systems easier to deploy and monitor. The integration approaches used to connect various business systems together, especially SOA approaches, also lend themselves to being utilized by deployment, configuration and monitoring tools.

However, in order to reap these benefits, the operations personnel who are designing and building the integration software between the primary systems must “design it like they plan to deploy it.” This means that the requirements for the overall system must not only be written in terms of important end-user functionality, but in terms of processes for deploying, configuring and monitoring performance of all systems involved. In some circumstances it can even be the case that these additional requirements will have an effect on how the end-user functionality is implemented, allowing for better auditing of activity in the system.

The author would like to report specific success with this approach, but is currently embarking on his first attempt to use it in a comprehensive manner. In his current position, a multi-year project is getting underway to upgrade or replace the majority of core systems operated by the I.T. organization. All of the approaches discussed in this paper are being used, with the intent to incrementally deploy a flexible system that can adapt quickly to changing needs and removes the “single-vendor handcuffs” that come along with pre-integrated systems.

The (Expected) Lesson: Place deployment, configuration and maintenance requirements at the same level of importance as user functionality requirements – the resulting ease of operations and stability of the system over time are just as important to end-user value.

Remember that change is your friend, not your enemy

The traditional 24x7 I.T. shop was typically a strong proponent of the status quo, preferring that the systems they were responsible for did not change frequently (or at all, if they could get away with it). This grew out of the fact that their success was most frequently measured in terms of stability, not in terms of innovation. The policies and procedures put in place were designed to ensure the highest levels of stability in systems, and frequently anticipated changes to those systems as rare anomalies.

Those days are gone, and operations professionals need to understand they are now expected to deliver value to the business through innovation just as much as their product development counterparts. Contemporary software development methodologies can be applied within IT shops and bring value both to the services they provide and to the operations professionals themselves. Being successful going forward will require that the policies, procedures and techniques employed in 24x7 I.T. shops embrace change and use it a central force to be successful in the eyes of their businesses.

References

- [1] – Wikipedia.com, “Fear, Uncertainty and Doubt”,
[http://en.wikipedia.org/wiki/Fear, uncertainty and doubt](http://en.wikipedia.org/wiki/Fear,_uncertainty_and_doubt), 7-Jul-2009
- [2] – Open Service Oriented Architecture collaboration, introductory material,
<http://www.osoa.org/display/Main/Home>, 30-Apr-2009
- [3] – OASIS, “OASIS Committees by Category: Web Services”, http://www.oasis-open.org/committees/tc_cat.php?cat=ws
- [4] – IBM Corporation, “video: Service Management vs. Systems Management”,
<http://www.youtube.com/watch?v=guHgiSOHImA>, 23-Apr-2009
- [5] – Wikipedia.com, “Software as a service”,
http://en.wikipedia.org/wiki/Software_as_a_service, 23-Jun-2009
- [6] – Wikipedia.com, “Free and open-source software”, <http://en.wikipedia.org/wiki/FOSS>,
6-Jul-2009
- [7] – Academic website for Barry Boehm at the USC Center for Software Engineering,
http://sunset.usc.edu/Research_Group/barry.html
- [8] – Wikipedia.com, “Barry Boehm”, http://en.wikipedia.org/wiki/Barry_Boehm, 19-Jun-1009
- [9] – Original Agile Manifesto, <http://agilemanifesto.org/>, 13-Feb-2001
- [10] – Object Management Group, “UML Resource Page”, <http://www.uml.org/>, 10-Mar-2009

Holding Our Feet to the Fire

Jim Brosseau
jim.brosseau@clarrus.com

Abstract

In recent times, there have been several movements in software development that appear to suggest that we can defer or eliminate many practices that apparently slow down our ability to develop our products.

As we have all seen in this industry, though, as movements such as Agile or Lean gain popularity, there is much that is lost in the translation to the masses, and these innocuous statements become embraced a little too tightly. There are claims of universality from all corners of the methodology debate, and that 'barely sufficient methodology' often becomes 'insufficient', with disappointing or disastrous results.

In this presentation, Jim suggests that a balance is required. There are some practices that need to be considered for any project, as the cost of deferring or ignoring them becomes extremely costly to the organization, and results in less delivered value. Balancing appropriate weighting of these practices with appropriate management of change becomes the optimal way of driving a project to successful completion.

Jim describes different types of practices to be considered, approaches to recognize reasonable application along the way, and identifies the practices that we absolutely need to move forward in the lifecycle to ensure that success means creation of the value we actually intended to deliver.

Biography

Jim has been in the software industry since 1980, in a variety of roles and responsibilities. He has worked in the QA role, and has acted as team lead, project manager, and director. Jim has wide experience project management, software estimation, quality assurance and testing, peer reviews, and requirements management. He has provided training and mentoring in all these areas, both publicly and onsite, with clients on three continents.

He has published numerous technical articles, and has presented at major conferences and local professional associations. His first book, *Software Teamwork: Taking Ownership for Success*, was published in 2007 by Addison-Wesley Professional. Jim lives with his wife and two children in Vancouver, Canada.

Introduction

In recent times, there have been several movements in software development that appear to suggest that we can defer or eliminate many practices that apparently slow down our ability to develop our products.

For example:

- Jim Highsmith has advocated a 'barely sufficient methodology' [1],
- Ken Schwaber and Mike Beedle suggest that 'Teams are put in a time box and told to create product', and assert that 'keeping models synchronized with code is a maintenance burden in any organization' [2].

As we have all seen in this industry, though, as movements such as Agile or Lean gain popularity, there is much that is lost in the translation to the masses, and these innocuous statements become embraced a little too tightly. There are claims of universality from all corners of the methodology debate [3], and 'barely sufficient methodology' often becomes 'insufficient' [4], with disappointing or disastrous results.

A balance is required. There are some practices that need to be considered for any project, as the cost of deferring or ignoring them becomes extremely costly to the organization, and results in less delivered value. Balancing appropriate weighting of these practices with appropriate management of change becomes the optimal way of driving a project to successful completion. The trick, of course, is to understand when to make those decisions.

When are we better served to hold our feet to the fire?

There are a couple of areas that lend themselves to conscious decisions early on, regardless of the project lifecycle, the size and shape of the team, or the relationship with the customer. While mature teams with depth of experience might apparently be able to breeze through these decisions quickly, it is still critical to ask ourselves if we have really nailed these elements before we proceed on any new project.

No decision or shared understanding should ever be considered cast in stone, regardless of when it is determined in the project lifecycle. Early agreements need to be seen as our best understanding at the time, and future events always involve the possibility of invalidating an early assumption. While some would argue this means deferral is better, these decisions described here, made early, serve to reduce the risk of conflict downstream by serving to align the stakeholders in the same direction.

Overall, there are five core elements that should be consciously considered on any project. While in many mature teams this consideration may be straightforward, to ignore these elements in the name of quickly creating product can have devastating results:

- Intentional management of the team dynamics,
- Alignment of the group with a shared vision,
- Explicitly identifying the value to be delivered on the project,
- Quantifying the quality of the product to be built, and
- Analyzing the system as extensively as possible.

The first element centers on how the people involved in the project will work together. In every group I work with, when asked about the characteristics of their best project experience, the vast majority of the responses center on relationships: good, positive communication, respect and trust are always evident.

While some teams may have the luck or serendipity to start out with strong relationships, most are well served to take the time to build and reinforce strong relationships between one another. Even for those that start out fine, the fact that push will usually become shove on projects means that the apparently strong relationships can quickly erode when the pressure rises. Team members need to understand each other's goals, needs and motivations, need to agree on mechanisms for resolving disagreements respectfully, need to understand how to best relate to one another.

The remainder of these elements critical to project success aligns the team to ensure everyone is working on the same project. A shared vision for the project, clear completion criteria in terms of value to be delivered to the customer and expression of scope that involves a quantified quality definition are critical before moving forward - think of this as an agreement on where the finish line actually is.

Note that this does not necessarily mean a comprehensive, exhaustive definition of scope. For those projects where there will be plenty of discovery downstream, we should still strive to identify where in the project we will be making these deferred decisions.

There is value in confirming our assumptions about what we believe we all know, and we need to know what we don't yet know.

Consciously Build Effective Teams

In working with a wide variety of teams in both industry and academia, there is often discussion of what needs to be captured in a persistent form. There is always resistance to documentation of any kind on projects, but there is one document that turns out to be the most relevant to the success for many teams. It is their team agreement.

For reasons that are not surprising, many people see team agreements as a waste of time, a useless document that is full of flowery language, and can be dismissed along with many vision statements that are found in lobbies around the world. Indeed, for some teams I work with, this viewpoint is not far from the truth. It is seen as an interesting diversion and generates some fun discussion about personalities, but it is not seen as the real work to do: the assignments due to be graded.

From my experience, though, there is a clear correlation between a respect for their team agreement and how well teams perform on their projects.

We use the Tuckman model for team development, which many people recognize as the Forming/Storming/Norming/Performing model. Usually, the teams progress through the team agreement with a false sense of security that they have quickly and painlessly managed to get to the performing stage as a team. That dreaded storming stage never seemed to appear at all.

What has actually happened, in most cases, is that the team has mistaken this initial team agreement as final, and any underlying issues tend to stay there, lurking, often brought to the surface later in the project.

This raises a couple of issues that teams need to address. It is clearly insufficient to hack together a team agreement and think the job is done. No team can assume they are faring well, that they are a well-performing team, until they have managed to get through a phase of conflict unscathed. Until then, and possibly afterwards as well, there remains a significant risk that the team can collapse with conflict.

If the team agreement is intended to help a team manage through crisis, there are a couple of things that should be discussed and agreed upon that are not often dealt with.

Feedback needs to be open and honest, of course, and team members need to be able to serve this up in a face-to-face manner. This feedback should include the strengths as well as the challenges, which should include some discussion of how these may be resolved. In other words, we need to avoid purely negative feedback, it needs to be provided with a goal of strengthening the team overall.

The discussions around the team agreement should include how to give reasonable feedback, but how to take feedback as well. Remember, the team agreement is simply the minutes of the important teaming discussions that you have had, not the production of a document. When we receive feedback, we need to see it as merely a data point. We can choose to be offended by this data, we can choose to 'shoot the messenger', or we can try to find the kernel within the data that supports improved relationships.

These discussions are never over at the beginning of the project when we have produced that document. In fact, if you think you have finished with your team agreement by producing a document rather than having the deep discussions, be prepared to deal with conflict sometime in the future, when your team really does run into trouble. This advice holds true for any document you produce, of course.

Work to a Compelling Vision

A critical component to sustained team success is finding a way of keeping everyone jazzed.

We all hear about amazing places to work, and Google comes up in many of these conversations. They have flexible work conditions, great perks, even one day a week where employees focus on whatever they want to, which has been the fruitful source of many of their products or research areas. Done right, everyone wins, and Google's success is at least partially a testament to this approach.

If we miss one component of what has been successful for others, though, things won't work out as we had hoped. In one shop I have worked with, they had set up what seemed to be a great work environment – totally flexible work hours, a comfortable workplace in one of the city's trendiest areas. They even provided the incentive to the troops to be creative and come up with great new ideas.

What happened was the opposite of what was intended. While not everyone took personal advantage of the flexibility, there was enough of a drop in energy that the net effect from the group was disappointing. Product wasn't coming through as expected. Few of those new ideas materialized. In the end, it turns out a couple of people were let go.

What went wrong? If we take a look at what Google stands for, there are a couple of statements that may be relevant. Buried deep in their Philosophy is the statement "Give the proper tools to a group of people who like to make a difference, and they will." While it sounds like this group did what they could to provide the proper tools and environment, the desired result didn't materialize. Granted, their pockets are not quite as deep as Google's, but could the challenge be in that phrase "who like to make a difference"?

Stop right there. I'm not suggesting that they were a bunch of deadbeat employees that don't like to make a difference. Intrinsically, we all want to make a difference in what we do, but we all also go through stages of lower energy.

More accurately, alongside the great work environment, there needs to be a fresh, exciting *raison d'être*, something that motivates and inspires us to greatness. The second relevant statement from Google's site is their mission statement itself: "to organize the world's information and make it universally accessible and useful".

Few companies have such a compelling vision. While I don't think we can all have such strong visions for our existence, we need to develop the best vision we can, and it needs to be something that the entire team buys into. It has to be something that jazzes everyone in the group, something that inspires them to make a difference. Whether it is inspiration to defeat a common enemy, or altruistic and world-saving, there should be no question about what you are all trying to achieve.

A good workplace is important, to be sure, but the motivation needs to be more than a request to get your work done. A strong vision can truly jazz a team, it is the difference between the workplace being a job and a joy.

Identify the Value to be Delivered

It can be difficult enough to get a project team to focus on delivery of value when we are starting a project; it is all that much tougher to remain focused on this prize as the project plays out. One of the main reasons for this is that the tools we use to manage projects tend to divert our focus elsewhere.

It is easy and commonplace to frame a project in terms of cost and schedule. These are two dimensions that we are all familiar with, and the traditional emphasis of these on projects reinforces these often shortsighted habits. While these are important, they pale in significance to the critical dimension of delivered value.

Yes, we want to complete the project on time and not lose our shirts doing so, and on some projects, these dimensions can be absolutely rigid. If we haven't made the stakeholders' world better in some way, if we have not delivered value, we have not done our job.

In working with project teams, setting expectations for value to be delivered can be one of the most difficult things we do. We need to do so as clearly and precisely as possible, which unfortunately requires more thinking and collaboration than most teams are accustomed to at this early stage.

We need to define value in terms that are relevant to stakeholders. Are we intending to reduce their costs by x% in the next 12 months? To increase market share or improve their quality of outputs or reduce their risk? We need to agree on expectations of how their lives will be better, and we need to be specific, measurable, and time based. All these objectives need to be relevant to the stakeholder needs, and they need to be attainable within our cost and time constraints.

This expressed value to be delivered is a precursor to capturing the user stories or traditional requirements of our systems, and serves as a basis to determine whether any of these user stories or requirements even make sense as part of the project: both initially, and for those proposed downstream.

The weaker we are at crisply defining expected value, the easier it is to fall astray as the project progresses. More importantly, the easier it is to artificially declare project success at the end of the day: "I managed to spend all your money in the time allotted!" can be all the good news we can muster on a project with a poorly structured expression of delivered value.

Even Earned Value analysis really has nothing to do with delivering value: it would be more aptly named Spent Budget analysis, as an entire project can progress very well by these standard reports, but fail to provide any value. The 'value' in Earned Value, is assumed to have been included in the definition of the activities to be performed, but is not actually measured: we accrue costs and effort expended, value is inferred. The same argument holds true for user stories.

We need to define value up front, to be sure, but we also need to remember that whenever we are thinking about progress, usually in terms of cost and time, we need to continuously ask ourselves: are we making progress towards our goal of providing value?

This is true for all of the activities we originally envisioned when we put our original schedule together, but it also remains important when we address change. Indeed, this becomes an important differentiator that allows us to manage change effectively: if a proposed change does nothing to maintain or improve the value delivered to the stakeholders, we have a strong argument for not including it in the project.

This alone is more than enough reason to invest the time to set expectations about delivered value on a project. The cost of the effort to do so will usually be more than made up if we can use it to reject a single proposed change that would otherwise diminish the value we deliver.

Quantifiably Specify Quality Needs

For all the companies I have worked with as an employee or as a consultant, whether they suggest they are agile or not, there have always been issues in the area of understanding the scope of work.

One of the recurring areas where many teams fail to specify the needs of what they are about to build or buy, is the area of quality. Everyone I have worked with suggests that quality is of utmost importance, yet most fail to precisely quantify what quality means in the context of their product.

There are many reasons for this deficiency. Some take the term quality to align with qualitative, and therefore don't grasp that these can and should be expressed in a quantified fashion. Unfortunately, suggesting that a system needs to be fast or user-friendly or secure just doesn't cut it when we are trying to test against these terms.

Others struggle to make the leap from the taxonomies that generally cover the landscape of quality, but are not easily expressible in quantified terms. It can take a great deal of time in the school of hard knocks to recognize that user-friendly needs to be expressed as a combination of a wide range of quantifiable ideas: acceptable response times and feedback, adherence to GUI standards, consistent error-handling mechanisms and a host of others.

Again, most teams don't do a good job in this area, and the results are often either accidental adequacy, or more often a trip back to the drawing board.

Within agile teams, there is a notion of an architectural spike: analyzing just enough of the necessary architecture of the system that will allow you to progress forward with the current set of stories so that the risk of major refactoring is minimized. This is a powerful concept that can work very well for the functional considerations of the system being built.

I would suggest that there is a sister spike that should take place around the same time: a quality spike can give us just enough insight to ensure that we are building to meet the quality needs of our system.

Indeed, if we perform a reasonable quality analysis at this point in the game, agile practices actually put us ahead of the curve when compared to traditional approaches. If maintainability is a quality attribute that is of interest to us, then the principles of a shared code base, ongoing refactoring and frequent releases play to our favor. These practices can be expressed as requirements that support our building a system that is maintainable.

Similarly, test-driven development and a unit test infrastructure support the testability of the system, as does the practice of frequent releases. Refactoring supports reusability, the list goes on. For those quality attributes that are of interest to the development organization, agile practices serve to inherently produce a higher quality system.

For those quality attributes that are of interest to the end-user, though, there is value in performing that analysis as a quality spike. As these attributes are strongly domain and application dependent, and will vary in importance dramatically, we cannot simply rest on our development practices to ensure that we meet these quality needs.

We need to sit down with our customer and decide which attributes are most important, or even relevant, and translate these into a series of measurable characteristics that we can then build to as we flesh out our stories. We do the best we can at this time, remembering that even if it is a spike, it can generate tremendous value.

Many of these attributes - such as security, usability, interoperability, and others - weave throughout all of the stories we are building, and consciously articulating these alongside our architectural spike gives us a much higher likelihood of efficiently meeting our goals. Indeed, some of these may be critical enough to drive stories of their own, stories we might otherwise only consider much later in the game.

Reasonably Analyze the System Early

Generally, when we talk about software testing, this is seen as the dynamic execution of the software system and peering into the code, rather than any other form of validating or verifying the system. Not static analysis of code, not peer reviews or inspections.

This trend of greater emphasis on this type of software testing is not good for the industry, and the fact that most companies I have worked with do not make a distinction between QA and testing compounds the problem.

Don't get me wrong; being effective at testing software products is not a bad thing. Indeed, given the way most companies develop software, we had better be good at finding all the bugs in our products. The challenge here is not testing; it is the emphasis on exhaustive testing over more efficient analysis techniques earlier in the lifecycle.

Let's pull out some stats from the industry. Numerous studies show that half or more of all defects found on software projects have their root in a requirements issue. One study at TRW [6] indicated 54% of all errors were found after unit testing, and 83% of these were requirements and design-based errors. Think critically about the defects that have been logged for your products, I would expect you to see a similar trend.

Layer on top of that a study with the US Navy [7] that indicated that incorrect facts accounted for 49% of requirements errors, and an additional 31% were errors of omissions. Having worked on defense projects as well as a wide range of projects in other sectors, I'd say that it is safe to assume that those 'errors of omission' could easily double in most shops.

One more data point that has been reflected in numerous studies, including one from Hewlett-Packard [8]: a requirements-based flaw can cost over 100x to fix after the system has been deployed, compared to the cost of fixing the problem closer to when it was injected into the system. The problem is, many shops first validate their systems only when the code is running.

The point is this. Throw all these studies together, and it becomes pretty clear that we spend way too much time cleaning up the spills that never should have happened in the first place.

I see a trend developing that more organizations are leaning on testing to actually figure out what their requirements should have been. In the recent past, the following points came out in discussion with various companies:

- One shop identified numerous defects for non-typical issues around data that should have been considered as alternatives and exceptions in their use-case analysis.
- In the same place, a tax lawyer indicated that she gets called into projects when a problem has been discovered, rather than identifying the relevant tax-related business rules up front and developing the system to accommodate them accordingly.
- Another shop found a particularly nasty bug where one data element was implemented in slightly different ways in different locations throughout the system. A data dictionary might have come in handy here.
- Then there is the company that only realized when integrating the whole system together that the system took minutes for what should have been a very fast response time. And the company that arrived at the integration stage to find they don't know how to test what they had built in the first place. Oops, maybe an analysis of quality attributes would have set reasonable expectations for the design.

It would be easy to come up with other examples for quite a while, and you would likely have stories to contribute as well. All of these issues identified above were discovered through dynamically executing the code, and all issues were fixed in the code. Until these points were raised in the context of requirements discussions, the leap had not even taken place that all of these were actually requirements-based defects in the first place.

It sadly reminds me of an old Calvin and Hobbes cartoon, where Calvin asks his dad how they determine the load limit on bridges. His dad replies that "they drive heavier and heavier trucks across the bridge, and when the bridge breaks, they weigh the truck, and that's the limit!" Absurd, but that's effectively the approach in many software shops today.

It is easy to see why testing is leaned on so heavily in the industry. Even the most novice person on the team can be seen as productive by poking at applications and quickly find issues with the system. Testing has traditionally been the point where we get the bugs out of the system, and it is the teams and individuals that magically pull a bug-ridden application to a state where it can be shipped that are often seen as heroes. In some ways, test-driven development takes this to the extreme: tests are crafted in advance of the code.

Isn't that what those requirements and design stages we rush through are supposed to accomplish? While I respect that good testing is a mature discipline and a skilled tester can wring out many of the most elusive defects, the point remains that everyone is better off if the defects weren't injected into the system to begin with.

The industry as a whole has far greater opportunities available in getting better at analysis, not in more testing or even more effective testing. We need to figure out our equivalent to those bridge load limits in advance, and build systems that meet these specs.

Good, effective analysis is a matter of selecting which techniques to use that will give us the most valuable perspectives of what we are trying to build. Use cases alone are insufficient, as are 3x5 index cards. Having a customer embedded with the team gives us timely opinions and answers, but does nothing to make us more effective at understanding whether we have appropriately analyzed the problem.

We need to pick the right techniques, as each distinct view peels another layer off the problem, and if sequenced correctly, no step is difficult on its own. The challenge is in picking those appropriate techniques, and very few groups consciously choose how they will attack the analysis challenge for each project. Almost all standardized approaches lean towards specific analysis techniques or deferral of these important decisions downstream, rather than ensuring the team is equipped with a wide range of techniques required to do effective analysis, and the judgment required to select the right ones for the problem at hand.

Actually, this higher level strategizing is a technique that effective testers do well, but in the context of exercising the product that has been built rather than proactively developing the product correctly in the first place. The skills and attitudes that have already been developed in effective testers are the same required for effective analysts, except that this need is in an underappreciated phase of building our products.

Make the shift to deciding what you need to build and how you will build it rather than discovering the omissions after the fact. You will need far fewer test resources or use them more efficiently and strategically earlier in the project, you will reduce risk and uncertainty on your projects, and you will ship better products faster.

Dangers in Taking Short-cuts

When you get right down to it, we spend a lot of time every day making decisions. From the clear decisions of how we are going to spend our time, whether we are going to do this or that, to decisions that are not as consciously performed, like how we'll make that first coffee in the morning or choosing our route home in the evening.

On projects, the decisions we make on a daily basis determine the outcome of the project. Some have more impact than others, of course, but like a butterfly flapping its wings in the Amazon, many of our decisions have far greater impact than we might think.

Even more than those small decisions we do make, though, it is the decisions we often neglect to make on our projects that can have the greatest impact at all. As we all know, it's unlikely that this impact will be positive.

Many of the decisions we don't make fall under the category of assumptions.

When I set a group loose on an exercise during training, I rarely get a question about my instructions, even after I raise this very point. Far more often, though, when it is time to debrief the exercise results, many of the questions I have to field are actually about details that weren't even understood up front. The group has forged ahead with implicit assumptions.

The irony that these people have forged ahead without first clarifying the requirements is not lost on me.

We all go through life, everyday, supported by a large number of assumptions. We develop routines, standard ways of doing things to reduce the number of decisions we need to make. While chances are that these decisions were appropriate as a basis for our routines when they were formed, there's a good chance we are running on a bunch of decisions that are now, over time, seriously flawed. It wouldn't make sense to always question everything we do, but at least once in a while, we should step back and consider if our assumptions are still valid.

On projects, one type of decision we don't make are the ones where we simply choose the first good looking option that comes to mind.

We take the first words out of our end-user's mouth as gospel rather than analyzing whether this is indeed an appropriate requirement. Our system design is usually not the one chosen after comparing a range of potential options and selecting the best, but more likely the first one that pops into our head. I would guess that few of us have given much thought as to whether a bubble sort or a selection sort would be more appropriate in a given situation after we had to answer that question on an assignment back in school.

We look back on these decisions and lament "why hadn't I thought of that?" It is almost always better to select from a range of possibilities that we have generated than to just go with the first idea we get that might work.

Another type of non-decision occurs when we delay making a decision until we have run out of options, and there is only one way left to deal with an issue. This is often forced on us during projects as we discover late in the project issues that were injected in the early stages, but have been lying hidden, typically because we haven't hunted them down.

If we find a requirements flaw early, we have a wide range of alternatives for solving the problem. When we catch the same problem a week before we ship, it's far too late to adjust the architecture to address the issue. Deferring decisions to the last possible moment is not an effective time management strategy.

Finally, there are those decisions that are made without the right people involved in the process. Requirements or User Stories written by an analyst on his own, without consulting the developers for feasibility, the testers for testability, or sometimes even the customer for...well, the information simply cannot address all those different perspectives if written in a vacuum. The apparent cost savings of not involving the right people becomes far more expensive as these people discover the flawed assumptions.

Flawed assumptions, taking the first thing that will work, running out of options by deciding too late, and not involving the right people in the decision-making process: each approach can be devastating to your project, and we are all guilty of these on occasion.

Building a Structure for Safe Deferral

If we consciously focus on effective team dynamics and a strong shared vision, and clearly identify the intended value to be delivered and use this to build the clearest possible understanding of the structure of the desired product, we get the best of both worlds: strong team coordination and the agility to provide the best possible solution to the customer. We develop an infrastructure that allows us to efficiently and effectively defer other decisions to that 'last possible moment'.

By reasonably holding our feet to the fire, by making the right decisions earlier in the lifecycle, we end up with toasty little toes. If we defer or avoid these decisions, we often find that the fire is still there, and there is a good chance we'll get burned.

References

- [1] Jim Highsmith, Agile Software Development Ecosystems, Addison-Wesley, 2002
- [2] Ken Schwaber, Mike Beedle, Agile Software Development with Scrum, Prentice Hall, 2001
- [3] Barry Boehm, Richard Turner, Balancing Agility and Discipline, Addison-Wesley, 2004
- [4] Alistair Cockburn, Agile Software Development, Addison-Wesley, 2002
- [5] Bruce Tuckman, Developmental Sequence in Small Groups, 1965
- [6] TRW study as reported in Alan M. Davis, Software Requirements: Objects, Functions and States
- [7] The US Navy's A-7E project, as reported in Alan M. Davis, Software Requirements: Objects, Functions and States
- [8] Robert Grady, "Applications of Software Measurement Conference," 1999

ProdTest: A Production Test Framework for SAAS Deployment at Salesforce.com

Bhavana Rehani
brehani@salesforce.com

Kei Tang
kei.tang@salesforce.com

Abstract

Salesforce.com is the market and technology leader in Software as a Service (SaaS) [1] and Platform as a Service (PaaS) [2]. SAAS can be defined as a model of software deployment where an application is delivered to customers via the Internet. The company's portfolio of Salesforce.com CRM applications has revolutionized the ways that customers manage and share business information over the Internet, while the company's Force.com PaaS enables customers, developers and partners to build powerful on-demand applications [3] for any business process. The SaaS and PaaS on-demand models present unique challenges during new feature deployment. Unlike on premise software, new features are rolled out to numerous customers at the same time. Also, this rollout happens within a planned maintenance window. This means the R&D group has a very short period to ensure that the new release is not going to adversely impact customers' critical business operations. In order to verify that the release is good to go, QA engineers run a suite of sanity tests[4] during the maintenance window. Previously, testing used to be a time consuming manual process involving numerous QA engineers. To reduce this effort, the QA organization has developed a new automation framework (ProdTest) for executing tests on the production environment. ProdTest was developed as a testing framework at Salesforce.com to run automated tests on production environments during releases. This framework allows engineers to write both API and UI tests. Since the initial rollout, ProdTest has gradually evolved into a complex and sophisticated tool for writing automated tests. It has also become an integral part of our deployment process and has provided both measurable and intangible benefits, such as more predictable test runtime, fewer people involved in a release and an increased confidence in the release decision. This paper discusses the development of ProdTest including the technical, process and people challenges faced while building it. We also discuss the benefits to the organization and provide an insight into our future direction.

Biography

Bhavana Rehani is a senior quality assurance engineer at Salesforce.com. She joined Salesforce.com in 2006. She has learned a great deal by working with various teams on automation initiatives within the organization. She loves discovering innovative ways to automate test cases that are normally thought of as un-automatable. She especially enjoys utilizing JavaScript for testing complex web UIs. Bhavana earned her Masters degree in Software Engineering from Carnegie Mellon in 2006. She has also worked in the field of online learning at CDAC-Mumbai, a scientific society of the Department of Information Technology, Govt of India, from 2000 to 2004. Bhavana's interests include test automation, mentorship, educational technology and web programming.

Mr. Tang has over a decade experience in the software industry working in the aerospace, financial and CRM sectors. He has extensive experience in test harness design and implementation, web services and API testing, as well as developing and applying methods for model-based approaches to support defect removal and test automation. Mr. Tang is currently a software quality assurance manager at Salesforce.com, the leading provider of cloud computing services. He oversees the quality for the next generation UI development. He earned a B.Sc. in Aerospace engineering from University of Michigan at Ann Arbor, and a MS in Aeronautic and Astronautic Engineering from MIT.

1 Introduction

Salesforce.com is a leader in Software as a Service (SaaS) for business applications and is an innovator for Platform as a Service (PaaS). With on-demand models, Salesforce.com takes care of the deployment, delivery and upgrade of the applications so customers can focus on their own business execution. The unique challenges the Salesforce.com R&D group faces in supporting the on-demand model is very different than those for on-premise based software. One important difference is that the deployment of new functionalities takes place during a short maintenance window for numerous customers at the same time. Salesforce.com delivers a major release approximately three times a year, which gets rolled out to all customers. Due to high impact, it is crucial that we take maximum care while testing every new release. During the release, we must make decisions to move ahead or to rollback at every stage in a very short period of time. In order to ensure that the release is good to go to production, we carry out sanity tests on the new release before giving the final go ahead. Testing the release on the production environment is crucial since the production environment brings about additional complexity. For example, a production system of our size, that services over 150 million requests per day, has numerous points of failure in subsystems like the cache server, search engine, database server, app servers and various batch processes. Although Salesforce.com has an extensive automated suite of tests that runs on our internal test environments, these tests cannot uncover issues specific to our production environment. For example, a search related test that passes on the internal environments can fail on production if the production search process is switched off or incorrectly configured. During a release, the production environment undergoes numerous configuration, code and database changes. As a result, there are hundreds of tests that we need to run on production before signing off on the deployment of the release. Previously, these tests were run manually and required every QA engineer to participate. This was not scalable as the number of production instances and features were growing rapidly. Our solution was to create a production test automation tool (ProdTest) to eliminate the manual testing effort.

The following sections describe the evolution of ProdTest, the challenges faced in its development and technology used. We also describe the benefits achieved and our future direction.

2 Problem description

2.1 Previous process

During deployment of major releases, QA engineers need to verify the sanity of features they own. Previously, this involved a long and laborious process of manually verifying the sanity test cases on production during the maintenance window. The following summarizes some of the challenges faced during release time:

1) Meeting the time pressure of our maintenance window:

One of the biggest challenges of the manual testing approach was to keep within the time allocated for the maintenance window. With many people running tests at different phases of the release, it took too much time to find the issues, thus leaving very little time to get them resolved. With ever increasing features and complexity, completing everything during the maintenance window became risky.

2) Requiring numerous engineers on call:

At least one QA representative from each functional area was required to be on duty during the release. This meant at least 40 QA engineers working late to ensure deployment success over several weekends; three to four times a year.

3) Scaling for new instances:

The number of production instances for Salesforce.com has been increasing rapidly. Although this has allowed us to better accommodate our growing customer base, it has also added to the QA verification effort. Since each instance requires repetition of the verification process, a manual approach is not scalable.

4) Avoiding human error:

The manual testing approach was more prone to error. A new QA engineer may miss a critical test case resulting in defects and customer issues, which was unacceptable because quality is the number one goal for our R&D organization.

5) Keeping track of the communication:

Tracking of the testing process was difficult because so much back and forth was required between the various teams. Thus, when real issues were uncovered, it took longer to resolve them.

6) Managing stress:

Downstream effects included high levels of tension among the team and the need to plan vacations so that they do not conflict with major releases, etc.

3 New process proposed

In order to overcome the problems faced during manual verification, the QA organization devised a new test automation initiative called ProdTest. ProdTest was conceived as an automation framework allowing QA engineers to automate their sanity test cases and run them on production. The goals, technical details, challenges and benefits of ProdTest are discussed below.

3.1 ProdTest goals

The aim of ProdTest was to create a production validation tool to execute a suite of sanity test cases against the on-demand production environment using external API and UI requests. The tool would allow us to:

- Always meet the maintenance window to help us maintain availability[5]
- Reduce the number of QA engineers needed
- Remove the need for repetitive tasks
- Expose bugs that tend to appear only in the complex production environment
- Report test results and failures
- Automate both API and UI tests

4 Building ProdTest

In order to resemble the production environment accurately, ProdTest was developed as a client application which utilizes Salesforce.com's public web service APIs. These web service APIs are designed for use by our customers and partners to build their custom integrations. [6] Using the public APIs helped us 'act like the customer' without requiring additional functionality from the app server.

4.1 Technology

We started building the ProdTest framework as follows:

- Designing and implementing an Eclipse-based tool and UI for running tests
- Integrating with the testing framework JUnit
- Integrating with Selenium, the JavaScript based UI test automation framework [7]
- Importing our API wsdl for use by our test code
- Creating test utilities
- Adding configuration files to identify production environments

4.2 Process

Agile processes such as a daily scrum are widely adopted in Salesforce.com[8], so we utilized scrum for the development of ProdTest. We formed a new scrum team, which had a backlog, tasks and sprints. The team consisted entirely of QA Engineers. It was a flexible member team, such that various people participated in sprints at different times, based on their availability. This allowed ProdTest to grow continually with inputs from numerous QA engineers.

4.3 People

ProdTest was adopted as an important initiative by our technology division. It received great backing by senior management, adopted as an organizational goal and its importance emphasized. Writing ProdTest tests was not only greatly encouraged by the managers, but writing prodtests was included as a mandatory step before release. This greatly helped QA engineers to adopt and enhance the framework quickly.

4.4 Evolution of ProdTest

Initially, ProdTest had a small set of features. In time, these evolved into a more complex and sophisticated test set. The challenges and limitations faced helped us customize ProdTest to best meet our needs. Some of these challenges and steps taken to overcome them are discussed in the next section.

5 Challenges and solutions

Time taken to run tests:

We quickly adopted ProdTest and the number of automated tests increased. However, due to the increasing number of tests, the test suite took longer and longer to complete. In order to decrease the time taken to run ProdTest, we've taken the following steps:

- Categorize tests so that the most important (high impact feature) tests run before the others. We introduced 5 categories
 - **DVT** - Data validation tests after the upgrade
 - **prebatch** - Sanity tests run before batch services are up,
 - **postbatch** - Sanity tests run before the site is internally up,
 - **postwww** - Sanity tests that run after the site is internally up, and
 - **extended** - More detailed tests that are run after site is live

Categorizing our tests into various stages ensured that we could communicate the issues found as early as possible before proceeding to the next stage of the release.

- Support multithreading in ProdTest so that multiple tests can be run simultaneously on the same machine. This was especially challenging for Selenium tests. However, multithreading helped us reduce the test run time by making use of multiprocessor systems. We also saved time of opening and closing browsers for each Selenium test.

- Within a couple of releases, we had a better idea about maintaining test data, structuring test code and categorizing tests. We documented and shared these best practices with test authors so that they can write more efficient tests.
- Pre-create and maintain test data to avoid creating it during tests.
- Conduct additional code reviews to eliminate or modify redundant or inefficient tests.

Lack of support for certain types of tests:

Certain tests that are more complex or highly dependent on application server code cannot be easily automated with the limited public API based functions available in ProdTest. For a very small subset of tests, we needed to implement some one of automated solutions that utilized some portions of our internal APIs too.

Maintaining test data:

The automated tests in ProdTest run on pre-determined sets of data in the production environments. Keeping the data up to date for every release was a cumbersome task. Eventually, we solved this problem by creating tools to minimize the effort in data configuration.

Categorizing tests correctly:

Proper categorizing helps to minimize run time but it is important to educate and inform test authors what these categories really mean. A number of tests used to be annotated as prebatch, postbatch and postwww but should have fallen in the extended category. A longer pre-live test run meant a longer wait for allowing the site to go live. Extended tests run after the site has gone live and help discover lower priority defects that can be fixed in minor releases.

Eliminating false failures:

One of the most important challenges faced is eliminating false failures. Failures may result due to various environmental or data related problems. Differentiating between false failures and real issues that cause a test to fail continues as an ongoing effort. In order to reduce this effort, we've emphasized the need for maintaining a clean state of test data by having tests clean up after themselves. To identify environment issues quickly, some of the tests fail with an appropriate message such as 'Ensure the ____ process is up'.

5.1 Training

ProdTest has been an ongoing effort to reduce the number of QA engineers working during a release. The team was able to create the initial ProdTest infrastructure within 3 months of dedicated effort, and as a result, we have eliminated all manual test effort. Now there are two testing roles to support the release: the test master and test owners. A test master is responsible for triggering the test runs during various stages, gathering test run results and coordinating with test owners for re-executing the failed tests. Test owners, representing various functional teams, are standing by to manually verify any failed test to ensure that the failure was legitimate. We organized several training sessions in the form of dry runs to familiarize everyone with the new process.

5.2 Best practices

In order to minimize test run time, the following best practices were shared with test authors:

1. Selenium test cases take longer to run (due to additional work in spawning browsers) so write API tests whenever possible.
2. Be conservative of what goes into Go Live Tests such as prebatch, postbatch and postwww since these categories need to have the shortest runs possible.
3. Avoid doing test data setup in your tests. Create required data in the test organization early, much before tests are run on it.
4. Spread out test cases to run on different sets of data to minimize interference.
5. Assign an owner for each set of data.
6. Categorize test cases into separate categories, since our design allows different categories to safely run in parallel.
7. Create duplicate sets of test data to ease quick troubleshooting of test failures. In this way, if one set of data becomes corrupt or lost, one can rely on the second set.

6 Benefits of ProdTest

We significantly reduced the number of QA engineers needed during the deployment window by eliminating manual verification. This was our biggest achievement.

We also removed a great deal of stress for engineers by eliminating the need to manually run all the sanity tests within a very short time frame. Instead of running the tests, a subset of the test owners can be in a stand-by mode and concentrate on resolving any issues found by ProdTest.

Another major benefit is that ProdTest provides better deployment test coverage. Various complex user scenarios can now be added into the test suite, which were previously impossible to run manually due to time considerations. The added coverage helps uncover bugs that are specific to the production environment. For example, we've quickly caught errors due to processes being switched off, virtual IP configuration errors, etc. Having an automated run provides a predictable runtime for the sanity tests, which becomes extremely important when planning a system downtime.

Since the development of ProdTest, we have used it in other ways that yield additional benefits that we did not anticipate at first. For example, ProdTest is now used for other minor releases including patch releases, releases for database upgrades, etc. Running ProdTest for these releases at no additional cost ensures the highest quality for all our releases. Also, we run ProdTest a few times daily on our internal testing environments. These internal automatic runs help validate that the changes checked in for new feature development have not caused regressions. This helps us continuously monitor the quality of our current release in development.

7 Future Direction

ProdTest has come a long way. Today, it has over 1700 tests. These helped us reduce our deployment time and resources, and identify critical issues quickly. However, as the number of tests and deployment instances grows, we hope to further reduce the run time for critical tests. We are investing in tools that automate test data setup. The QA organization aims to achieve a very high level of dependability for ProdTest, such that we do not encounter any false failures due to environment issues, data issues, etc. This should help the QA organization reach our ultimate goal of being able to hand ProdTest over to the technical operations team, who can use it during deployment without any involvement from QA.

8 Conclusion

ProdTest serves as an excellent example of the strength of test automation. Today, almost all software companies invest resources towards software automation. ProdTest demonstrates how we can extend automation into different environments, which were previously not easy to test. Tools like ProdTest can bring about improvements in both process and product quality if designed and planned for correctly.

Aside from the intended benefits discussed in the paper, the ProdTest initiative helped us adopt and improve a slightly different type of agile development process. The ProdTest scrum team has been a flexible team consisting of several in and out members working at different times, based on their time availability. In spite of this, we have learned to take on new ProdTest commitments, task ownership, add requirements, and manage the product backlog in an effective and direct manner. This not only paves the way for future ProdTest related innovation and lays the foundation for kicking off other similar initiatives within the QA organization.

References:

- [1] Software as a Service, http://en.wikipedia.org/wiki/Software_as_a_Service
- [2] Platform as a Service, http://en.wikipedia.org/wiki/Platform_as_a_service
- [3] On demand software, http://www.webopedia.com/TERM/O/on_demand_software_delivery.html
- [4] Sanity testing, http://en.wikipedia.org/wiki/Sanity_testing
- [5] Trust.salesforce.com for Salesforce.com instance availability <http://trust.salesforce.com/>
- [6] Salesforce.com Web Services API http://wiki.apexdevnet.com/index.php/Web_Services_API
- [7] Selenium -Web Application Testing System <http://selenium.openga.org/>
- [8] Babinet and Ramanathan, Dependency Management in a Large Agile Environment, Agile 2008 Conference, Aug 2008

